Parallel and Distributed Systems for Probabilistic Reasoning

Joseph Gonzalez

December 2012
CMU-ML-12-111

MACHINE LEARNING
DEPARTMENT

Carnegie Mellon

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **DEC 2012**	2. REPORT TYPE	3. DATES COVERED **00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE **Parallel and Distributed Systems for Probabilistic Reasoning**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Machine Learning Department,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Scalable probabilistic reasoning is the key to unlocking the full potential of the age of big data. From untangling the biological processes that govern cancer to effectively targeting products and advertisements, probabilistic reasoning is how we make sense of noisy data and turn information into understanding and action. Unfortunately, the algorithms and tools for sophisticated structured probabilistic reasoning were developed for the sequential Von Neumann architecture and have therefore been unable to scale with big data. In this thesis we propose a simple set of design principles to guide the development of new parallel and distributed algorithms and systems for scalable probabilistic reasoning. We then apply these design principles to develop a series of new algorithms for inference in probabilistic graphical models and derive theoretical tools to characterize the parallel properties of statistical inference. We implement and assess the efficiency and scalability of the new inference algorithms in the multicore and distributed settings demonstrating the substantial gains from applying the thesis methodology to real-world probabilistic reasoning. Based on the lessons learned in statistical inference we introduce the GraphLab parallel abstraction which generalizes the thesis methodology and enable the rapid development of new efficient and scalable parallel and distributed algorithms for probabilistic reasoning. We demonstrate how the GraphLab abstraction can be used to rapidly develop new scalable algorithms for probabilistic reasoning and assess their performance on real-world problems in both the multicore and distributed settings. Finally, we identify a unique challenge associated with the underlying graphical structure in a wide range of probabilistic reasoning tasks. To address this challenge we introduce PowerGraph which refines the GraphLab abstraction and achieves orders of magnitude improvements in performance relative to existing systems.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT **Same as Report (SAR)**	18. NUMBER OF PAGES **181**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

Parallel and Distributed Systems for Probabilistic Reasoning

Joseph Gonzalez

December 2012
CMU-ML-12-111

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Carlos Guestrin, Chair
Guy Blelloch, CMU
David O'Hallaron, CMU
Alex Smola, Google
Jeff Bilmes, UW

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by AT&T Labs Fellowship Program, the ONR Young Investigator Program grant N00014-08-1-0752, the ARO under MURI W911NF0810242, DARPA IPTO FA8750-09-1-0141, the ONR PECASE-N00014-10-1-0672, the NSF under the Graduate Research Fellowship Program and grants NeTS-NOSS CNS-0625518, NeTS-NBD CNS-0721591, and IIS-0803333, and Intel Science and Technology Center for Cloud Computing.

Abstract

Scalable probabilistic reasoning is the key to unlocking the full potential of the age of big data. From untangling the biological processes that govern cancer to effectively targeting products and advertisements, probabilistic reasoning is how we make sense of noisy data and turn information into understanding and action. Unfortunately, the algorithms and tools for sophisticated structured probabilistic reasoning were developed for the sequential Von Neumann architecture and have therefore been unable to scale with big data. In this thesis we propose a simple set of design principles to guide the development of new parallel and distributed algorithms and systems for scalable probabilistic reasoning. We then apply these design principles to develop a series of new algorithms for inference in probabilistic graphical models and derive theoretical tools to characterize the parallel properties of statistical inference. We implement and assess the efficiency and scalability of the new inference algorithms in the multicore and distributed settings demonstrating the substantial gains from applying the thesis methodology to real-world probabilistic reasoning.

Based on the lessons learned in statistical inference we introduce the GraphLab parallel abstraction which generalizes the thesis methodology and enable the rapid development of new efficient and scalable parallel and distributed algorithms for probabilistic reasoning. We demonstrate how the GraphLab abstraction can be used to rapidly develop new scalable algorithms for probabilistic reasoning and assess their performance on real-world problems in both the multicore and distributed settings. Finally, we identify a unique challenge associated with the underlying graphical structure in a wide range of probabilistic reasoning tasks. To address this challenge we introduce PowerGraph which refines the GraphLab abstraction and achieves orders of magnitude improvements in performance relative to existing systems.

Acknowledgments

Research is a team effort and I was fortunate enough to be a part of an amazing team. I would like to thank my advisor Carlos Guestrin, who helped me focus on the important problems, guided me through the challenges of research, and taught me how to more effectively teach and communicate ideas both in writing and in presentations. In addition, Carlos gave me the opportunity to work with, learn from, and lead an exceptional team.

Much of the work in this thesis was done with Yucheng Low, who taught me a lot about systems, software engineering, and how to persevere through challenging bugs and complicated and even impossible proofs. Our many long discussions shaped both the key principles in this thesis as well as their execution. In addition, Yucheng was instrumental in developing many of the systems and theoretical techniques used to evaluate the ideas in this thesis. Finally, Yucheng's exceptional skills as a world class barista made possible many late nights of successful research.

Early in my graduate work at CMU I had the opportunity to work with Andreas Krause on Gaussian process models for signal quality estimation in wireless sensor networks. Andreas showed me how to apply the scientific method to design effective experiments, isolate bugs, and understand complex processes. Around the same time I also started to work with David O'Hallaron. As I transitioned my focus to the work in this thesis, David provided early guidance on scalable algorithm and system design and research focus. In addition, David introduced me to standard techniques in scientific computing and helped me build collaborations with Intel research.

The GraphLab project was made possible by an exceptional team which included: Yucheng Low, Danny Bickson, Aapo Kyrola, Haijie Gu, Joseph M. Hellerstein, Alex Smola, and Guy Blelloch. Yucheng worked closely with me to formulate and refine the abstractions, build the systems, and lead the GraphLab team. Danny Bickson's mastery of collaborative filtering, Gaussian BP, and community building fueled the excitement and adoption of the GraphLab project. Aapo Kyrola challenged our assumptions and helped us refine and simplify the GraphLab system and which ultimately lead an entirely new system. I would also like to thank Guy Blelloch for his early guidance designing and implementing GraphLab abstraction. Haijie Gu led the development of the PowerGraph partitioning system and graph representation and was instrumental in the success of the PowerGraph project. Joseph Hellerstein helped us focus the GraphLab abstraction and effectively present GraphLab to the systems community. Alex helped identify the high fan-in/out limitations of the GraphLab abstraction and was instrumental in the design and implementation of PowerGraph. I would also like to thank Alex for his guidance in my research career.

The work on parallel Gibbs sampling was done in collaboration with Arthur Gretton and later Emily Fox. Both Arthur and Emily helped me gain a better understanding of the work in sampling community and helped me master the analytical and algorithmic tools needed to develop sophisticated distributed Gibbs samplers.

I would also like to thank my thesis committee and the many members of the Select Lab for their guidance and feedback throughout my PhD. I would like to thank Steven Phillips who was as an external advisor and helped me develop an appreciation for the impact of research on the greater scientific community.

Finally, I would like to thank my family for always supporting and believing in me. I would especially like to thank my loving wife Sue Ann Hong whose patience, support, and guidance made this thesis possible.

Contents

1 Introduction 1

1.1 Example: Modeling Political Bias in a Social Network . . . 3

1.2 Contributions and Greater Impact . . . 4

1.3 Outline and Summary of Key Results . . . 5

2 Background 11

2.1 Probabilistic Reasoning . . . 11

2.1.1 Factorized Joint Probability Distributions . . . 12

2.1.2 Probabilistic Graphical Models . . . 13

2.1.3 Inference . . . 15

2.1.4 Parameter Learning . . . 16

2.1.5 Structure Learning . . . 16

2.1.6 Summary of Important Graphical Models Properties in the Parallel Setting . . . 16

2.2 Parallel and Distributed Systems . . . 17

2.2.1 Shared Memory . . . 17

2.2.2 Distributed Memory and Cluster Computing . . . 18

3 Probabilistic Inference: Belief Propagation 20

3.1 Belief Propagation . . . 20

3.1.1 The Message Schedule . . . 22

3.1.2 Opportunities for Parallelism in Belief Propagation . . . 22

3.2 Synchronous BP . . . 23

3.2.1 Synchronous BP in MapReduce . . . 24

3.2.2 Synchronous BP Runtime Analysis . . . 24

3.2.3 Analysis of Synchronous Belief Propagation on Chains . . . 25

3.3 The τ_ϵ Approximation . . . 26

3.3.1 Graphical Interpretation . . . 27

3.3.2 Empirical Analysis of τ_ϵ . . . 28

3.4 Round Robin Belief Propagation . . . 31

3.5 Wildfire Belief Propagation . . . 32

3.6 Residual Belief Propagation . . . 33

3.6.1 Limitations of Message Residuals . . . 35

3.6.2 Belief Residuals . . . 36

3.7 Splash Belief Propagation . . . 38

3.7.1 Optimal Parallel Scheduling on Chains . . . 38

3.7.2 The Splash Operation . . . 40

3.7.3 The Sequential SplashBP Algorithm 42
3.7.4 Evaluating SplashBP in the Sequential Setting 43
3.7.5 Parallelizing the Splash Algorithm 44
3.7.6 Parallel Splash Optimality on Chains 45
3.7.7 Dynamically Tuning the Splash Operation 48
3.7.8 Implementing SplashBP in the Shared Memory Setting 49
3.8 Belief Propagation Algorithms on Distributed Architectures 51
3.8.1 Partitioning the Factor Graph and Messages 52
3.8.2 Distributing Scheduling 56
3.8.3 Distributed Termination 57
3.8.4 The Distributed Splash Algorithm 58
3.8.5 Algorithm Comparison in the Distributed Setting 60
3.9 Additional Related Work 61
3.10 Conclusion 63

4 Probabilistic Inference: Gibbs Sampling 64
4.1 The Gibbs Sampler 65
4.2 The Synchronous Gibbs Sampler 65
4.3 The Chromatic Sampler 67
4.3.1 Properties of 2-Colorable Models 68
4.4 Asynchronous Gibbs Sampler 69
4.5 The Splash Gibbs Sampler 70
4.5.1 Parallel Splash Generation 71
4.5.2 Parallel Splash Sampling 74
4.5.3 Adaptive Splash Generation 75
4.5.4 A Need for Vanishing Adaptation 76
4.6 Experiments 81
4.7 Additional Related Work 82
4.8 Conclusion 83

5 GraphLab: Generalizing the GrAD Methodology 84
5.1 Introduction 84
5.2 Common Patterns in MLDM 86
5.2.1 Sparse Graph Structured Dependencies 87
5.2.2 Asynchronous Iterative Computation 87
5.2.3 Dynamic Computation: 89
5.3 High-Level Abstractions 89
5.3.1 Data-Parallel Abstractions 90
5.3.2 Process-Parallel Abstractions 91
5.3.3 Graph-Parallel Abstractions 92
5.4 The GraphLab Abstraction 93
5.4.1 Data Graph 94
5.4.2 GraphLab Vertex Programs 95
5.4.3 The GraphLab Execution Model 96
5.4.4 Scheduling Vertex Programs 97
5.4.5 Ensuring Serializable Execution 98
5.4.6 Aggregation Framework 100

5.5 Multicore Design 100
5.6 Multicore GraphLab Evaluation 101
5.6.1 Parameter Learning and Inference in a Markov Random Field 101
5.6.2 Gibbs Sampling 102
5.6.3 Named Entity Recognition (NER) 104
5.6.4 Lasso 105
5.6.5 Compressed Sensing 107
5.7 Distributed GraphLab Design 108
5.7.1 The Distributed Data Graph 109
5.7.2 Distributed GraphLab Engines 111
5.7.3 Fault Tolerance 113
5.7.4 System Design 115
5.8 Distributed GraphLab Evaluation 116
5.8.1 Netflix Movie Recommendation 117
5.8.2 Video Co-segmentation (CoSeg) 119
5.8.3 Named Entity Recognition (NER) 120
5.8.4 EC2 Cost evaluation 121
5.9 Additional Related Work 121
5.10 Conclusion 122

6 PowerGraph: Scaling GraphLab to the Web and Beyond 125
6.1 Introduction 125
6.2 Graph-Parallel Abstractions 126
6.2.1 Pregel 126
6.2.2 GraphLab 127
6.2.3 GAS Decomposition 127
6.3 Challenges of Natural Graphs 128
6.4 PowerGraph Abstraction 129
6.4.1 GAS Vertex-Programs 130
6.4.2 Delta Caching 132
6.4.3 Initiating Future Computation 132
6.4.4 Comparison with GraphLab / Pregel 133
6.5 Distributed Graph Placement 134
6.5.1 Balanced p-way Vertex-Cut 134
6.5.2 Greedy Vertex-Cuts 137
6.6 Abstraction Comparison 139
6.6.1 Computation Imbalance 140
6.6.2 Communication Imbalance 141
6.6.3 Runtime Comparison 141
6.7 Implementation and Evaluation 141
6.7.1 Graph Loading and Placement 142
6.7.2 Synchronous Engine (Sync) 143
6.7.3 Asynchronous Engine (Async) 144
6.7.4 Async. Serializable Engine (Async+S) 144
6.7.5 Fault Tolerance 145
6.7.6 MLDM Applications 145
6.8 Additional Related Work 146

6.9 Conclusion 147

7 Conclusion and Observations 150

7.1 Modeling Through Recurrence Relationships 151

7.2 Tradeoff between Parallelism and Convergence 152

7.3 Approximations and Parallelism 153

7.4 Challenges of Dynamic Scheduling 153

7.5 Importance of High-Level Abstractions 154

8 Future Work 156

8.1 Scalable *Online* Probabilistic Reasoning 156

8.2 Learning a Scalable Parsimonious Structures 157

8.3 Generalizing Prioritized Computation 158

8.4 Dynamic Work Balancing 158

8.5 Declarative Abstractions for Recurrence Resolution 159

8.6 The GrAD Methodology and General Graph Computation 159

9 Summary of Notation and Terminology 160

9.1 Terminology 160

9.2 Notation 161

Bibliography 163

Chapter 1

Introduction

Probabilistic reasoning forms the foundation of modern statistics and machine learning. From estimating the chance of rain to predicting a user's shopping interests we rely on probabilistic reasoning to quantify uncertainty, model noisy data, and understand complex phenomena. Probabilistic graphical models are one of the most influential and widely used techniques for complex probabilistic reasoning. By providing a compact representation of complex phenomena as well as a set algorithmic and analytical tools, probabilistic graphical models have been used to successfully untangle the structure [Yanover and Weiss, 2002, Yanover et al., 2007] and interactions [Jaimovich et al., 2006] of proteins, decipher the meaning of written language [Blei et al., 2003, Richardson and Domingos, 2006], and label our visual world [Li, 1995].

Historically, exponential gains in sequential processor technology have enabled increasingly sophisticated probabilistic graphical models to be applied to a wide range of challenging real-world problems. However, recent physical and economic limitations have forced computer architectures away from exponential sequential performance scaling and towards parallel performance scaling [Asanovic et al., 2006]. Meanwhile, improvements in processor virtualization, networking, and data-storage, and the desire to catalog every fact and human experience, has lead to the development of new hardware paradigms like cloud computing and massively parallel data-centers which fundamentally challenge the future of sequential algorithms [Armbrust et al., 2010].

Unfortunately, the algorithms and techniques used to analyze and apply probabilistic graphical models were developed and studied in the sequential Von Neumann setting. As a consequence, recent efforts [Chu et al., 2006, Panda et al., 2009, Wolfe et al., 2008, Ye et al., 2009] in scalable machine learning have overlooked these rich models and techniques in favor of simpler unstructured models and approximations which are more immediately amenable to existing large-scale parallel tools, abstractions, and infrastructure. Alternatively, when confronted with computationally challenging graphical models we typically resort to model simplification rather than leveraging ubiquitous and increasingly powerful parallel hardware.

In this thesis we introduce the simple GrAD design methodology which leads to a new set of efficient algorithms, theoretical tools, and systems for scalable parallel and distributed probabilistic reasoning. The **Gr**aphical, **A**synchronous, **D**ynamic, (**GrAD**), design methodology is composed of the following three basic principles:

1. **Graphical**: Explicitly represent statistical and computational dependencies as a sparse graph. The sparse graphical representation enables the parallel decomposition of algorithms and exposes the data locality needed for efficient distributed computation.

2. **Asynchronous**: Allow computation to proceed asynchronously as dependencies are satisfied obeying the dependency structure encoded in the graph. By allowing computation to run asynchronously as dependencies are satisfied we can reduce wasted computation and build more efficient parallel and distributed systems. By obeying the dependency structure we can obtain provably correct asynchronous algorithms.

3. **Dynamic**: Adaptively identify and prioritize sub-problems that when solved make the most progress towards the solution. By first identifying and solving the key sub-problems we can reduce the overall work and more effectively use computational resources.

The GrAD design methodology lead to the following core principle evaluated in this work:

Thesis Statement: By factoring large-scale probabilistic computation with respect to the underlying *graphical structure*, *asynchronously* addressing sequential dependencies, and *dynamically* scheduling computation, we can construct efficient parallel and distributed systems capable of scalable probabilistic reasoning.

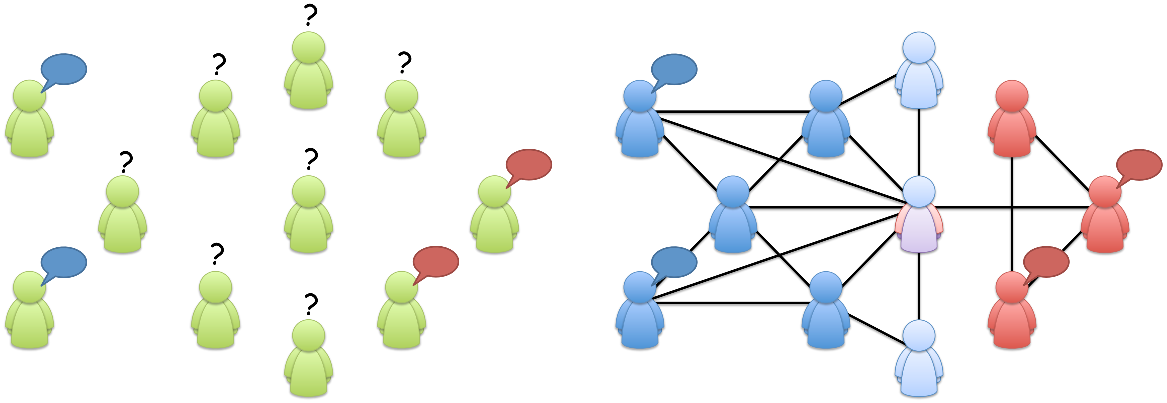
We evaluate the thesis statement by applying the GrAD design methodology to two of the most widely used probabilistic inference techniques, loopy belief propagation (Chapter 3) and Gibbs sampling (Chapter 4), and derive a sequence of efficient parallel algorithms for probabilistic reasoning. In the process, we develop basic theoretical tools like the τ_ϵ-approximation Section 3.3 and the connection between graph coloring and Gibbs sampling Section 4.3 to characterize the efficiency, scalability, and correctness of parallel inference algorithms. Using large-scale real-world problems we experimentally evaluate the resulting algorithms in the parallel and distributed settings. We find that by applying the GrAD methodology we are able to obtain provably efficient and correct parallel and distributed algorithms that substantially outperform existing approaches.

By generalizing our work on probabilistic inference algorithms, we derive the GraphLab high-level graph-parallel abstraction (Chapter 5) which explicitly targets the GrAD design principles while providing strong serializability guarantees (Section 5.4.5) that simplify the design and implementation GrAD algorithms. We use the GraphLab abstraction to implement our graphical model inference algorithms as well as several new algorithms demonstrating the applicability of the GrAD methodology to a wider class of problems (e.g., Co-EM in Section 5.8.3 and the Lasso in Section 5.6.4). We implement the GraphLab system in both the multicore and distributed settings generalizing our earlier system design work by introducing new dynamic schedulers, improved distributed graph representations, and fault-tolerance. Finally, we evaluate our implementation of the GraphLab system and show orders of magnitude improvements over traditional data-parallel abstractions like Hadoop.

From our experience in designing and deploying GraphLab, we identified a critical challenge in scaling graphical computation to massive real-world graphs. The power-law degree distribution, a defining characteristic of a vast majority of real-world graphs, violates several of the key assumptions that form the foundation of the GraphLab system as well as other widely used systems for graphical computation. To address this challenge, we introduce PowerGraph which refines the GraphLab abstraction and *exploits* the power-law structure as well as a common pattern in GrAD algorithms. As a consequence PowerGraph is able to substantially reduce communication overhead and introduce a new approach to distributed graph representation. We demonstrate that the PowerGraph system both theoretically and experimentally outperforms existing graph computation systems (including GraphLab) often by orders-of-magnitude.

1.1 Example: Modeling Political Bias in a Social Network

Suppose we are interested in estimating the political bias (e.g., liberal or conservative) of each user in a large social network with hundreds of millions of users and billions of relationships. We might begin by building a simple classifier that predicts political bias of a user based on the presence of politically polarizing text in her profile. Applying this simple classifier to each user in isolation is a well studied problem [Chu et al., 2006] and can easily scale to hundreds of millions of users by applying classic parallel computing techniques and abstractions [Dean and Ghemawat, 2004]. While we may be able to accurately estimate the political bias of users that frequently post polarizing comments, for the vast majority of users who post infrequently the profile may not contain sufficient information to draw strong conclusions about political bias (see Figure 1.1a).



(a) Without Social Network

(b) Predictions With Social Network

Figure 1.1: **Social Network Example:** Consider the task estimating the political bias of users in a large social network. In **(a)** we have politically informative comments for only a small subset of the users leaving us unable to make predictions about the vast majority. However by including the social network structure **(b)** we can substantially improve our predictions.

Graphical: If we rely exclusively on the profile text to estimate political bias, we are discarding the import *relationships* between users. Moreover, knowing something about the political bias of your friends, their friends, and your greater social community can reveal a lot about your political bias (see Figure 1.1b). Unfortunately, directly modeling all of these relationship leads to overly parameterized models and is generally computation intractable. Instead, we can decompose the relationship along the original social *graph*. However, to be able to model the long-range relationships (e.g., the effect of your community on your political bias) we will need to *iteratively* move *information* through the graph (transitive closure). This can be accomplished by using inference algorithms that *factorize* over the graph structure (i.e., operate only at the level of each vertex and its neighbors) and iteratively refine simple models at each vertex based on the models on neighboring vertices.

Asynchronous: Having decomposed the task of modeling long range relationships into the task of iteratively refining local relationships, we introduced a substantial amount of parallelism. For example in the large social network we could in principle refine the model for each user *simultaneously*. As a consequence, many have proposed *synchronously* refining all users models by simulating a separate processor for each user. However, we will show that it is substantially more efficient to refine users profiles *asynchronously* as processor resources become available and using the most recent estimates for neighboring profiles. Intuitively, if a processor is currently refining one of your neighbors models then it is often better to wait for your neighbors new model then to recompute based on the old model.

Dynamic: In a large social network there are often *trend setters* whose opinion shapes their community. Intuitively, applying computational resources to refine the local models for the *trend setters* can have a greater impact than applying the same resources to the vast majority of the remaining users. Alternatively, accurately estimating the models for the vast majority based on inaccurate models for the trend setters wastes resources since the estimates for the vast majority are likely to change as we refine the estimates for the trend setters. By *dynamically* identifying and focusing computational resources on the trend setters we can eliminate wasted computation and accelerate convergence.

GraphLab: In order to implement and run the proposed *Gr*aphical *A*synchronous *D*ynamic (GrAD) algorithm we would need to address a wide range of complex multicore and distributed system design challenges including: efficient re-entrant data structure design, race and deadlock elimination, distributed data placement and coordination, and fault-tolerance. While there are a several high-level parallel and distributed abstractions which effectively hide the challenges of system design, none of them naturally express GrAD algorithms. To fill this void we introduced the GraphLab abstraction which directly targets GrAD algorithms and hides all of the challenges of multicore and distributed system design. Using the GraphLab abstraction, we can quickly and easily implement and debug our GrAD algorithm. Then using the many interchangeable GraphLab schedulers we can evaluate the effect of different dynamic schedules on convergence and scalability. Finally, we can apply our algorithm to large graphs in both the multicore and distributed settings.

PowerGraph: However, if we try to scale GraphLab to the full social network with billions of edges using a large cluster of machines, we quickly find that communication overhead dominates and performance degrades. Furthermore if we try to use other abstractions or traditional techniques for implementing distributed graph computation the situation does not improve. The problem is that the power-law sparsity structure, commonly found in graphs derived from natural phenomena (e.g., social networks and the web), violates many of the assumptions made by traditional graph-parallel systems (including the GraphLab system). However, by exploiting a common pattern in GrAD algorithms and the structure of power-law graphs we can refine the GraphLab abstraction and produce a new system PowerGraph that is orders of magnitude faster and more efficient.

1.2 Contributions and Greater Impact

We evaluate the thesis statement by developing the core algorithms, theory, and tools needed to scale probabilistic reasoning to challenging real-world problems in the parallel and distributed settings. More precisely, we:

- design and implement a suite of efficient parallel and distributed graphical model inference algorithms based on the GrAD principles. Chapter 3, Chapter 4)
- develop theoretical tools to characterize the efficiency and scalability of the proposed algorithms as well as the underlying parallel structure of inference in graphical models. (Chapter 3, Chapter 4)
- introduce the GraphLab abstraction and system which simplifies the design and implementation of scalable GrAD parallel and distributed algorithms. (Chapter 5)

- introduce the PowerGraph abstraction and system which refines GraphLab to efficiently express computation on power-law graphs and capture the fundamental structure of the GrAD methodology. (Chapter 6)
- evaluate the proposed techniques on real-world problems using a variety of parallel platforms and demonstrate substantially improved scaling, computational efficiency, accuracy, and speed relative to existing systems and techniques.

The algorithms, systems, and abstractions developed throughout this thesis have already started to have significant impact on the greater research community. The GraphLab and PowerGraph systems along with open-source implementations of a wide variety of GrAD algorithms for scalable probabilistic reasoning have been widely downloaded and used to address tasks ranging from collaborative filtering to natural language modeling. Along with help from collaborators we have fostered a community around graphical approaches to scalable probabilistic reasoning.

In the future, we believe the emphasis on graph structured models and systems for scalable efficient inference, will enable the development and application of substantially richer models to the massive scale real-world problems. Throughout this thesis we give examples of sophisticated structured models and how they are able to address complex real-world problems. By efficiently using parallel and distributed resources, the algorithms and techniques described in this thesis can substantially decrease the time required to design, evaluate, and debug new sophisticated models on the *complete data* allowing model designers to capture the full complexity of the underlying phenomenon. We believe this will lead to a proliferation of new structured models and inference techniques in a wide range of applications.

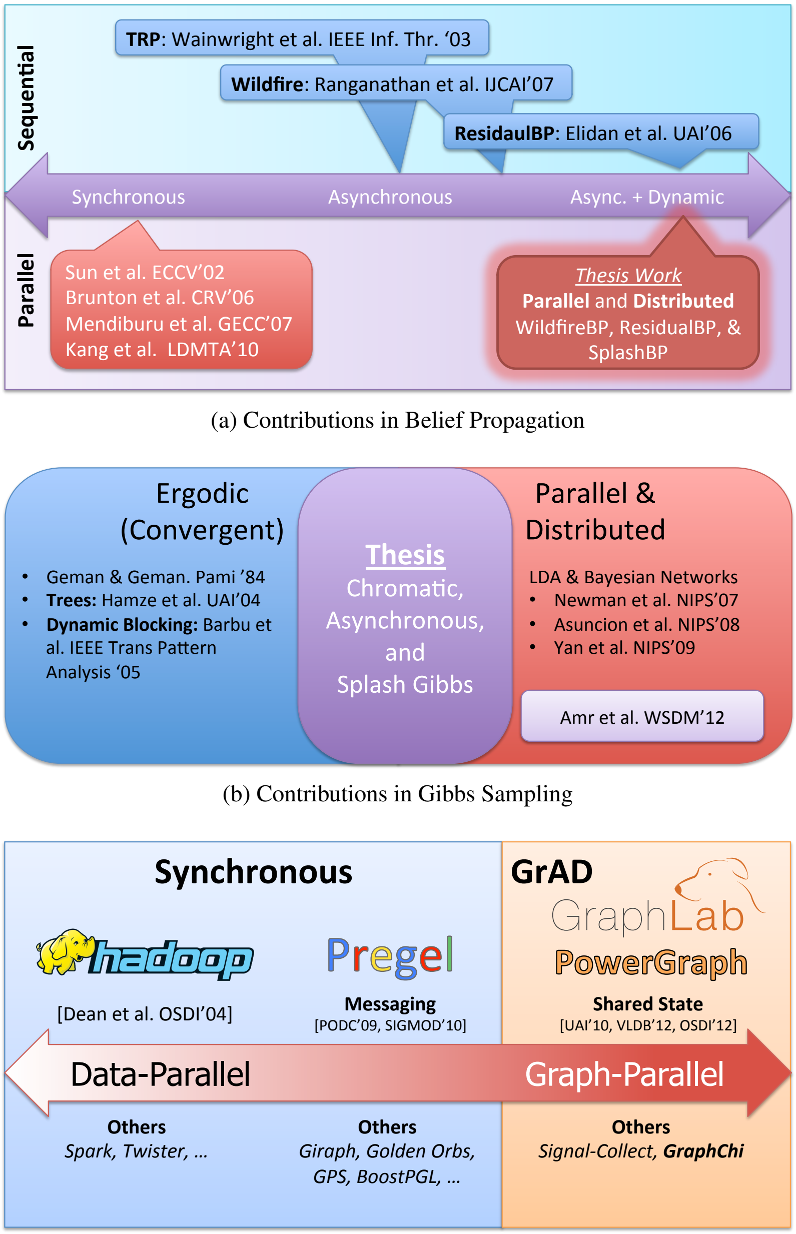
The GraphLab and PowerGraph high-level abstractions will allow advances in scalable system design to be directly applicable to algorithms built upon these abstractions. By abstracting away the complexity of scalable probabilistic reasoning, the GraphLab and PowerGraph abstractions enable system designers to develop and apply advances in eventually consistent data representations, parallel and distributed data structure design, and networking to a wide range of probabilistic reasoning tasks. Similarly, by isolating the challenges of scalable parallel and distributed system design, GraphLab and PowerGraph system enables experts in probabilistic reasoning to easily and efficiently construct rich models and algorithms which leverage a wide range of architectures.

Finally, we believe the GrAD design methodology and the lessons learned in this thesis will help refocus the course of future research in scalable probabilistic reasoning away from simple unstructured models and parallelism at the expense algorithm efficiency and towards rich structured models and efficient GrAD algorithms and systems. This thesis provides a tractable approach to rich structured models with scalable and efficient distributed algorithms and systems and opens a wide range of new research directions including, distributed dynamic scheduling, the role of model structure in approximate probabilistic reasoning, scalable asynchronous consistency protocols, efficient distributed placement and representation of natural graphs, and new approaches to recursive model design.

1.3 Outline and Summary of Key Results

We begin in Chapter 2 by briefly reviewing some of the key terminology, operations, and properties of probabilistic graphical models as well as parallel and distributed algorithms and systems.

In Chapter 3 and we apply the GrAD methodology to design and implement algorithms and systems for

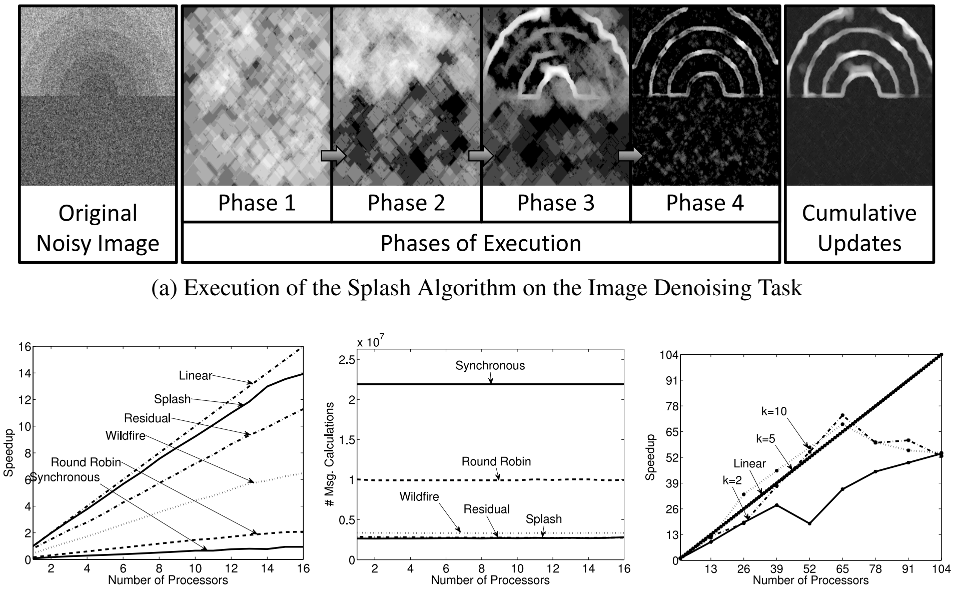


(a) Contributions in Belief Propagation

(b) Contributions in Gibbs Sampling

(c) Contributions in High-Level Abstractions and Systems

Figure 1.2: The contributions of the thesis work in the context of existing work in graphical models inference (belief propagation and Gibbs sampling) as well high-level abstractions and systems for parallel and distributed computation.



(a) Execution of the Splash Algorithm on the Image Denoising Task

(b) Speedup

(c) Work

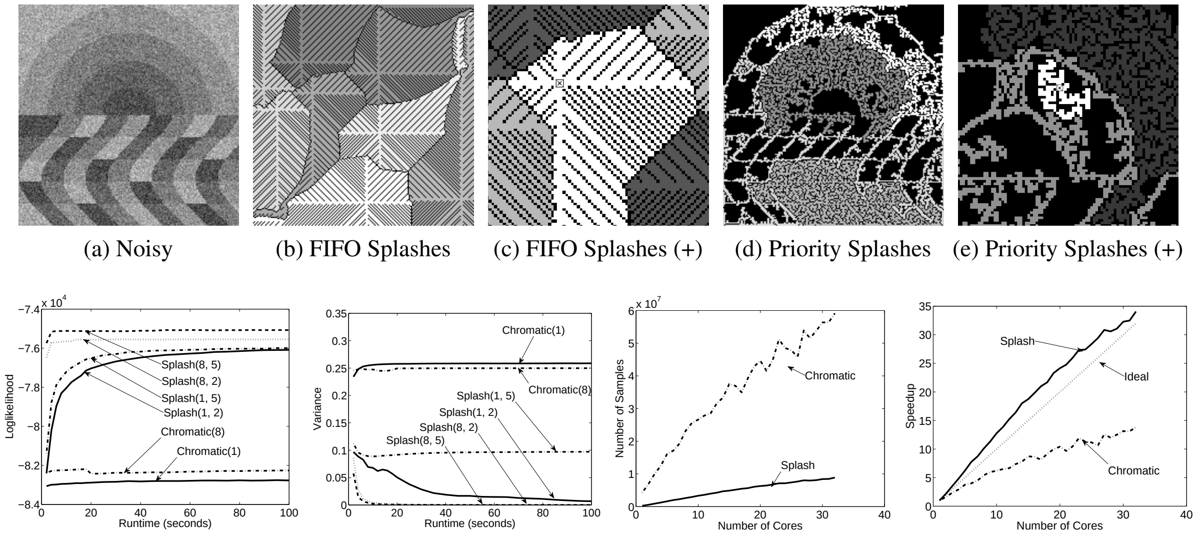
(d) Speedup

Figure 1.3: **Key Results from Work on Belief Propagation**: In **(a)** we plot the number of times each pixel is recomputed when applying the SplashBP algorithm to the image denoising task. Brighter pixels have been recomputed more often. As the algorithm converges it *adaptively* identifies the challenging parts of the image and focuses computation where it is needed. **(b,c)** By introducing more aspects of the GrAD we are able to improve the scalability and efficiency of probabilistic reasoning systems. The Wildfire, Residual, and Splash algorithms implement all three of the GrAD principles. By closely preserving the optimal sequential schedule the SplashBP algorithm is able to most efficiently schedule computation. **(d)** To deal with work imbalance in the distributed setting we introduced a simple over partitioning heuristic (larger k) which can substantially improve scalability.

probabilistic inference using loopy belief propagation. We incrementally introduce each aspect of the GrAD approach and derive a sequence of algorithms which build upon each other (see Figure 1.3). In the process we develop basic theoretical tools for reasoning about the efficiency and convergence of parallel belief propagation algorithms and relate the parallel scaling to both structural properties of the graph as well as underlying *model parameters*. In the end we arrive at the SplashBP algorithm which adaptively moves wavefronts across the graph and embodies the key ideas described in the thesis statement. We then implement the each algorithms in the multicore and distributed settings and evaluate their performance on a wide range of metrics using real-world problems.

In Chapter 4 we apply the GrAD methodology to Gibbs sampling, a widely used inference technique in Bayesian statistics. As with our work on belief propagation we derive a sequence of algorithms each introducing new properties of the GrAD methodology. In contrast to existing work on parallel Gibbs sampling, we prove that the resulting parallel algorithms are ergodic and therefore preserve the same guarantees as their sequential counterparts. In addition we relate the parallel scaling to graph theoretic properties of the underlying Markov Random Field and reveal the limiting distribution of existing non-ergodic parallel Gibbs samplers. Finally, we arrive at the SplashGibbs algorithm which applies many of the lessons learned from our work on SplashBP to substantially accelerate burn-in and mixing (see Figure 1.4. We then implement each of the algorithms in the multicore setting and evaluate their performance on a collection of synthetic and real-world sampling tasks.

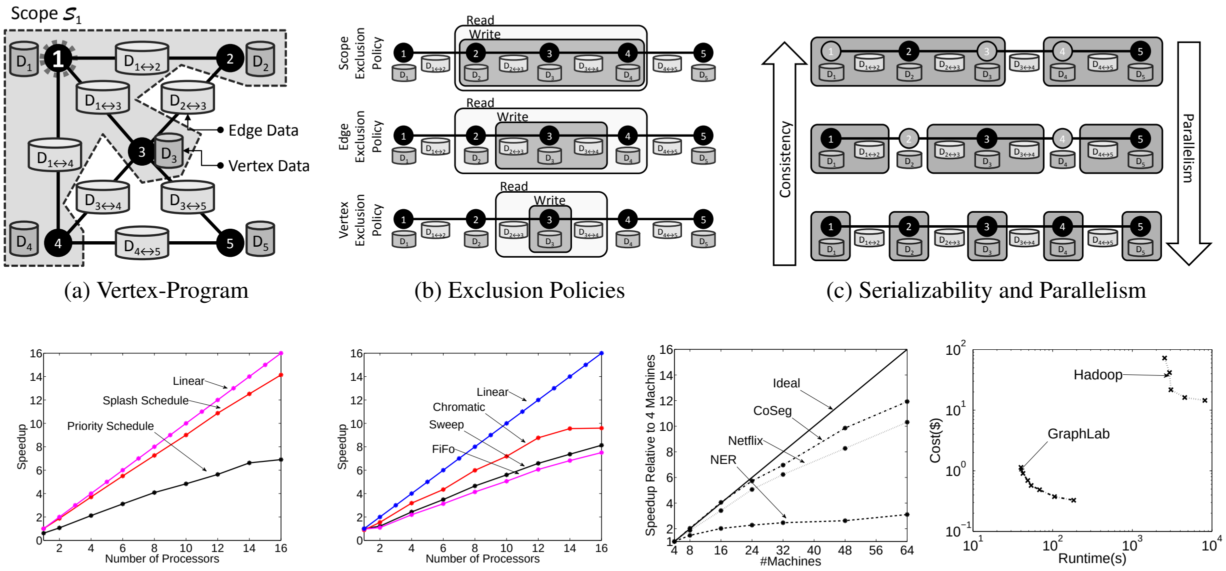
In Chapter 5 we introduce the GraphLab graph-parallel abstraction which generalizes our work in graphical model inference and is specifically targeted at algorithms that follow the GrAD methodology. We begin by



(a) Noisy (b) FIFO Splashes (c) FIFO Splashes (+) (d) Priority Splashes (e) Priority Splashes (+)

(f) CORA-1 Likelihood (g) CORA-1 Variance (h) CORA-1 Samples Count (i) CORA-1 Speedup

Figure 1.4: **Key Results from Work on Gibbs Sampling**: **(a)** A noisy image of a sunset illustration. **(b - e)** Different thin junction tree Splashes constructed for an image denoising task (grid MRF). Each splash is shown in a different shade of gray and the black pixels are not assigned to any Splash. In **(c)** and (e) we zoom in on the Splashes to illustrate their structure and the black pixels along the boundary needed to maintain conditional independence. **(f-i)** Comparison of Chromatic sampler and the Splash sampler at different settings on a real world citation modeling task. **(f)** The un-normalized log-likelihood plotted as a function of running-time. **(g)** The variance in the estimator of the expected assignment computed across 10 independent chains with random starting points. **(h)** The total number of variables sampled in a 20 second window plotted as a function of the number of cores. **(i)** The speedup in number of samples drawn as a function of the number of processors.



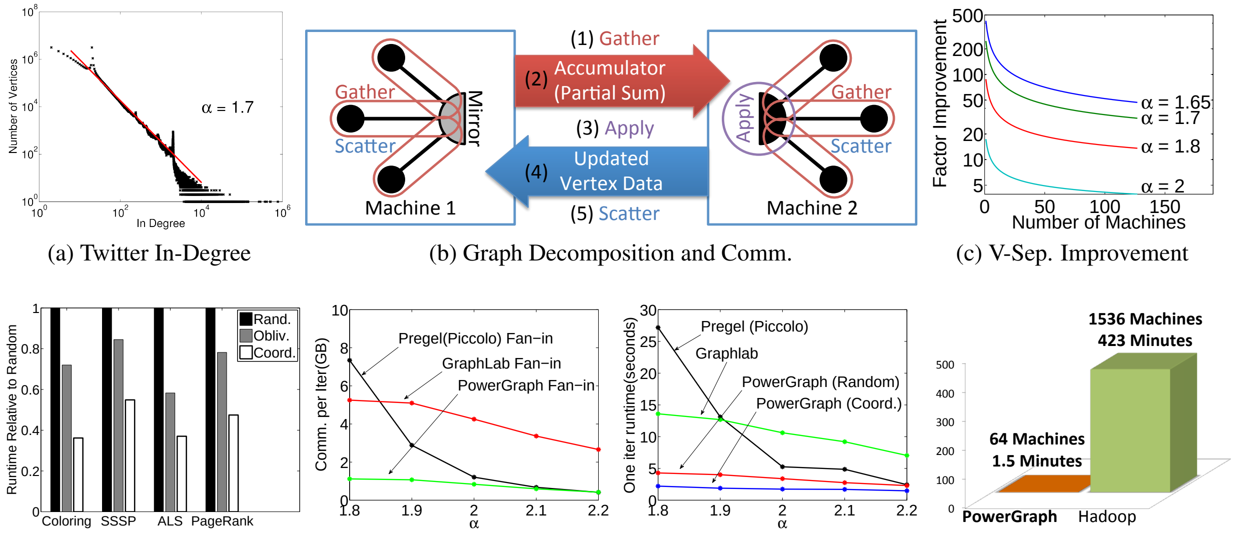
(a) Vertex-Program (b) Exclusion Policies (c) Serializability and Parallelism

(d) BP Speedup (e) Gibbs Speedup (f) Other Alg. Speedup (g) EC2 Price/Performance

Figure 1.5: **Key Results for GraphLab**: Computation in GraphLab is encoded as a vertex programs that can read and modify the data on adjacent vertices and edges. In **(a)** the vertex-program running on the vertex 1 has access to all the data in the shaded region. GraphLab automatically enforces strong serializability guarantees through three exclusion policies **(b)** which allow the user to explore the tradeoff **(c)** between parallelism and data consistency. In **(d,e)** we plot the multicore speedup of our earlier parallel belief propagation and Gibbs sampling algorithms running within the GraphLab framework. In **(f)** we plot the distributed speedup of several new GrAD algorithms implemented using the GraphLab abstraction. Finally, in **(g)** we demonstrate the substantial financial savings and performance gains of the GraphLab abstraction when compared to Hadoop.

reviewing the key properties of algorithms that follow the GrAD methodology and discussing how existing high-level abstractions are not well suited for these algorithms. We then present the GraphLab abstraction and demonstrate how GraphLab is able to naturally represent GrAD algorithms while providing strong serializability guarantees. We then implement and evaluate the GraphLab system in the multicore and distributed settings demonstrating strong scalability and substantial gains relative to MapReduce based abstractions (i.e., Hadoop).

In Chapter 6, we identify the power-law sparsity structure found in most real-world graphs (Figure 1.6) as one of the critical challenges to the scalability of distributed graph-parallel computation. We describe how the high-degree vertices and low quality balanced cuts found in power-law graphs leads to substantial storage and communication overhead for GraphLab as well as other popular message based graph-parallel abstractions. We then propose the GAS decomposition which factors vertex programs over edges in the graph and enables the use of vertex cuts to efficiently place large power-law graphs in a distributed environment. We then propose a set of three simple distributed techniques to construct vertex cuts as the graph is loaded and theoretically analyze these techniques demonstrating order of magnitude gains over standard randomized edge cuts. Finally, we evaluate the PowerGraph system on several real-world problems and demonstrate order of magnitude gains against existing state-of-the-art results.



(a) Twitter In-Degree (b) Graph Decomposition and Comm. (c) V-Sep. Improvement

(d) Effect of Partitioning (e) Power-law Fan-Out Comm. (f) Power-law Fan-Out Runtime (g) Triangle Count Runtime

Figure 1.6: **Key Results for PowerGraph**: The vast majority of real-world graphs have a **(a)** power-law degree distribution. Here we have plotted the in-degree distribution for the twitter follower graph. The power-law degree distribution leads to a small set of very high degree vertices. The PowerGraph abstraction **(b)** splits high degree vertices across machines and factor vertex-programs along edges in the graph using the Gather-Apply-Scatter (GAS) decomposition. The PowerGraph abstraction enables the use of vertex-cuts which substantially reduce the communication overhead **(c)** relative to traditional edge-cuts. We introduced several new partitioning heuristics **(d)** which improve the overall performance algorithms run on the PowerGraph system. In figures **(c,d)** we demonstrate that the PowerGraph abstraction out performs GraphLab and Pregel and is robust to changes in the power-law constant (smaller α implies more high degree vertices). Finally, in **(g)** we plot the runtime in minutes of PowerGraph and Hadoop when computing triangles in the twitter follower graph. PowerGraph is over 282 times faster than the best published result for this problem.

Chapter 2

Background

This work combines ideas from statistical machine learning, parallel algorithms, and distributed systems. In this chapter we provide the background on the relevant concepts from each of these fields and introduce some the notation used throughout the remaining chapters.

2.1 Probabilistic Reasoning

Suppose we are interested in modeling the political bias (e.g., liberal or conservative) of users in a large social network. We might begin by representing the political bias of each user i as a Bernoulli random variable $X_i \sim \text{Bern}(\theta_1)$ taking the value 0 if i is a liberal and 1 if i is a conservative. The parameter θ_1 determines the probability $\mathbf{P}(X_i = 1) = \theta_1$ that any user is conservative and can be estimated by polling or expert knowledge (e.g., $\theta_1 = 0.5$). While this simple model provides a prediction of the political bias for each user, it does not take into account information unique to each user (e.g., the users profile or purchase history).

We can improve our model by incorporating the observed profile text y_i for each user. This can be accomplished by defining the **conditional probability** $\mathbf{P}(Y_i \mid X_i)$ of observing the text Y_i given the user has the political bias X_i. Using expert knowledge or polling data for our users we can estimate $\mathbf{P}(Y_i \mid X_i)$

$\mathbf{P}(Y_i = y_i \mid X_i = x_i) =$

	$x_i = 0$ (liberal)	$x_i = 1$ (conservative)
”Obamacare” $\notin y_i$	0.8	0.3
”Obamacare” $\in y_i$	0.2	0.7

(2.1)

Using Bayes rule we can combine the **prior** $\mathbf{P}(X_i)$ with the **likelihood** $\mathbf{P}(Y_i \mid X_i)$ to compute the **posterior** probability that a particular user is a conservative or liberal given their profile text:

$$\mathbf{P}(X_i | Y_i) \propto \frac{\mathbf{P}(Y_i \mid X_i)\,\mathbf{P}(X_i)}{\sum_{X_i} \mathbf{P}(Y_i \mid X_i)\,\mathbf{P}(X_i)} \tag{2.2}$$

While we may be able to accurately estimate the political bias of users that frequently post polarizing comments, for many users who post infrequently the profile may not contain sufficient information to draw strong conclusions about political bias.

If we rely exclusively on the profile text to estimate political bias, we are discarding the import *relationships* between users. Moreover, our model assume that knowing the political bias of a user's friends tells us *nothing* about the users own political bias, which is clearly not the case. More precisely, our model incorrectly assumes that the random variables corresponding to political bias are *independent*. However relaxing the independence assumption in the context of large datasets is both challenging and one of the principle goals of this thesis work.

We can relax the independence assumption by defining the full **joint probability** over all the random variables in our model. In the context of the social networking example, if we just considered the random variables X_i the joint probability takes the form:

$$\mathbf{P}\left(X_1, \ldots, X_n\right) \tag{2.3}$$

which returns a probability for every possible assignment to each random variable X_i in the model. For a social network with only two people we could encode the joint probability as a simple table:

$$\mathbf{P}\left(X_1, X_2\right) = \begin{array}{|c|c|} \hline & \mathbf{P}\left(x_1, x_2\right) \\ \hline x_1 = 0, x_2 = 0 & .4 \\ x_1 = 0, x_2 = 1 & .1 \\ x_1 = 1, x_2 = 0 & .1 \\ x_1 = 1, x_2 = 1 & .4 \\ \hline \end{array} \tag{2.4}$$

which effectively states that friends are more likely to share the same political view. Using the joint probability function we can answer questions about the relationship between random variables. For example, in the above two user social network the probability that user one is a liberal given that user two is a liberal can be expressed as:

$$\mathbf{P}\left(X_1 = 0 \,|\, X_2 = 0\right) = \frac{\mathbf{P}\left(X_1 = 0, X_2 = 0\right)}{\sum_{x_1} \mathbf{P}\left(X_1 = 0, X_2 = 0\right)} = \frac{0.4}{0.4 + 0.1} = \frac{4}{5} \tag{2.5}$$

However, to extend this approach to even a small social network of 100 users we would need a table with 2^{100} rows and the calculation in Eq. (2.5) would requiring a summation over 2^{99} states!

2.1.1 Factorized Joint Probability Distributions

Directly learning and manipulating the full joint probability distribution over every possible states for all random variables is typically intractable. However, for most problems we can *factorize* the joint probability distribution:

$$\mathbf{P}\left(X_1, \ldots, X_n \,|\, \theta\right) = \frac{1}{Z(\theta)} \prod_{\mathbf{A} \in \mathcal{F}} f_{\mathbf{A}}(\mathbf{x}_{\mathbf{A}} \,|\, \theta), \tag{2.6}$$

into a products of functions $f_{\mathbf{A}}$ called **factors**. Each factor $f_{\mathbf{A}}$ is positive function, $f_{\mathbf{A}} : \mathbf{x}_{\mathbf{A}} \to \mathbb{R}^+$, over a small subset $\mathbf{A} \subseteq \{1, \ldots, n\}$ of random variables $\mathbf{X}_{\mathbf{A}}$. The factors are parametrized by a set of parameters θ, and the **partition function** $Z(\theta)$ is defined as:

$$Z(\theta) = \sum_{X_1} \cdots \sum_{X_n} \prod_{\mathbf{A} \in \mathcal{F}} f_{\mathbf{A}}(\mathbf{x}_{\mathbf{A}} \,|\, \theta). \tag{2.7}$$

While we will primarily focus on distributions over discrete random variables $X_i \in \mathcal{X}_i = \{1, \ldots, |\mathcal{X}_i|\}$ most of the proposed methods may also be applied to continuous distributions.

If we return to our social networking example, one possible factorized representation of the joint probability of all the users and their profiles is:

$$\mathbf{P}\left(X_1, \ldots, X_n, Y_1, \ldots, Y_n\right) \propto \left(\prod_{i=1}^{n} \mathbf{P}\left(Y_i \,|\, X_i\right)\right) \left(\prod_{i=1}^{n} \mathbf{P}\left(X_i\right)\right) \tag{2.8}$$

This factorized representation is equivalent to the *independent* user model we proposed earlier. Another possible representation for the joint probability distribution is:

$$\mathbf{P}\left(X_1, \ldots, X_n, Y_1, \ldots, Y_n\right) \propto \left(\prod_{i=1}^{n} \mathbf{P}\left(Y_i \,|\, X_i\right)\right) \left(\prod_{(i,j) \in E} f_{\{i,j\}}\left(X_i, X_j \,|\, \theta\right)\right). \tag{2.9}$$

where E is the set of all edges in the social network and the the factors $f_{\{i,j\}}$:

$$f_{\{i,j\}}\left(X_i, X_j \,|\, \theta\right) = \begin{cases} 1, & \text{if } x_i = x_j \\ \theta, & \text{otherwise} \end{cases} \tag{2.10}$$

(where $0 < \theta < 1$) are defined over friends i and j and encode the prior belief that friends should share a similar political bias. Given this model we can now condition on the observed user text and answer questions about each user's political bias in the context of their friends and their greater community.

In this example we started by modeling users independently and then improved our model by capturing the relationships between friends. In the process we described a compact way to express a complex joint distribution. We now formalize this modeling technique and describe the methods for applying and analyzing these models.

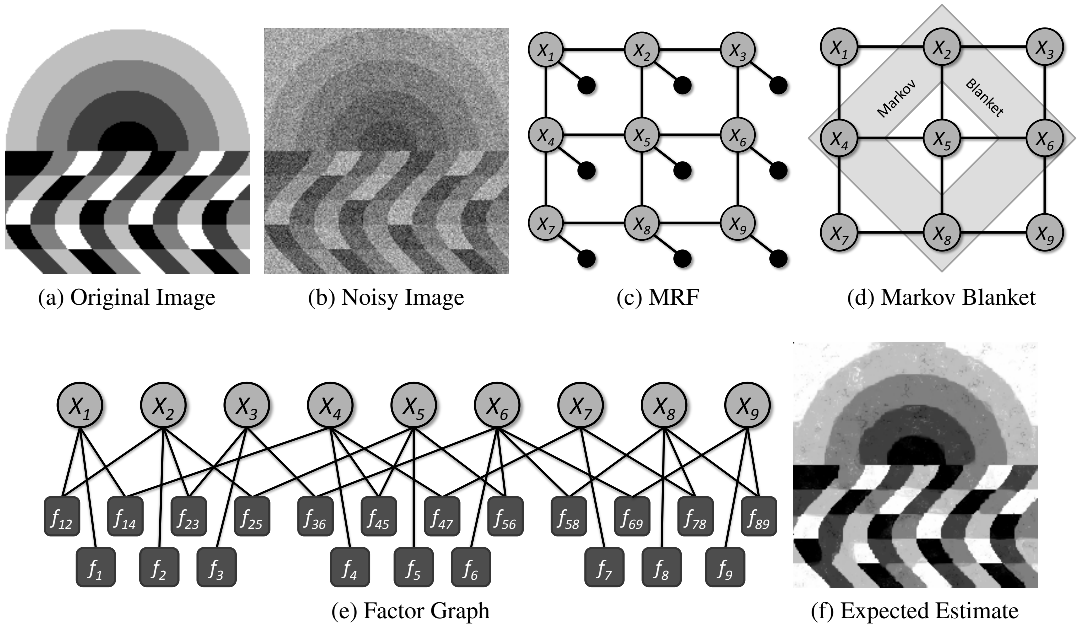
2.1.2 Probabilistic Graphical Models

Probabilistic graphical models provide a common language and set of algorithmic tools for efficiently representing and manipulating large factorized distributions. This is accomplished by constructing a correspondence between the factorized distribution and a combinatorial graph representation upon which we can define algorithms for learning and manipulating the underlying model.

A **Markov Random Field** (MRF) is an undirected graphical representation of a factorized distribution. The MRF corresponding to the factorized probability distribution in Eq. (2.6) is an undirected graph over the variables where X_i is connected to X_j if there is a $\mathbf{A} \in \mathcal{F}$ such that $i, j \in \mathbf{A}$. We denote the set of indicies corresponding to the variables neighboring X_i as $\mathcal{N}_i \subseteq \{1, \ldots, n\}$ and therefore $\mathbf{X}_{\mathcal{N}_i}$ is the set of neighboring variables.

To provide a easy to visualize illustration of a Markov Random Field we introduce the digital image denoising task. We can construct a factorized distribution to model the noisy pixels in a digital image (see Figure 2.1). For each pixel sensor reading we define an observed random variable y_i and a corresponding unobserved random variable X_i which represents the true pixel value. We can then introduce set of factors corresponding to the noise process (e.g., Guassian):

$$f_{\{i\}}(X_i = x_i \,|\, Y_i = y_i, \theta) = \exp\left(-\left(\frac{(x_i - y_i)^2}{2\theta_1^2}\right)\right). \tag{2.11}$$



(a) Original Image (b) Noisy Image (c) MRF (d) Markov Blanket

(e) Factor Graph (f) Expected Estimate

Figure 2.1: Image denoising problem. **(a)** The synthetic noiseless original image. **(b)** The noisy pixel values observed by the camera. **(c)** The MRF corresponding to the potentials in Eq. (2.11) and Eq. (2.12). The large gray vertices correspond to the latent unobserved pixel values X_i while the dark black circles correspond to the observed pixel values y_i **(d)** The Markov blanket for variable X_5. **(e)** The denoise factor graph. As with the MRF we have removed the Y_i random variables and instead associated each X_i with the unary factor f_i. **(f)** The expected assignment to each of the latent pixels.

To encode smoothness in the image we introduce factors which encode similarity between neighboring pixels (e.g., Laplace smoothing):

$$f_{\{i,j\}}(X_i = x_i, X_j = x_j \,|\, \theta) = \exp\left(-\theta_2 \left|x_i - x_j\right|\right). \tag{2.12}$$

for adjacent X_i and X_j (e.g., cardinal neighbors). Then we can express the factorized joint distribution (Eq. (2.6)) as:

$$\mathbf{P}\left(X_1, \ldots, X_n, Y_1, \ldots, Y_n\right) \propto \left(\prod_{i=1}^{n} f_{\{i\}}(X_i = x_i \,|\, Y_i = y_i, \theta)\right) \left(\prod_{(i,j)\in E} f_{\{i,j\}}(X_i = x_i, X_j = x_j \,|\, \theta)\right). \tag{2.13}$$

The Markov random field for the image denoising problem is drawn in Figure 2.1c, and the indicies of the variables neighboring X_5 in Figure 2.1c are $\mathcal{N}_5 = \{2, 4, 6, 8\}$. To resolve the underlying image w e might then want to compute the expected assignment $\mathbf{E}\left[X_i\right]$ for each pixel as in Figure 2.1f.

The edges in the MRF encode both the statistical dependencies between the variables and the *computational dependencies* of the algorithms that operate on these models. As a consequence the MRF is a crucial structure in designing scalable parallel and distributed algorithms for structured probabilistic reasoning. In addition, the MRF also provides a means to leverage the substantial existing work in parallel graph algorithms and job scheduling.

When building large-scale probabilistic graphical models, sparsity in the MRF is a desirable property.

Densely connected models typically require too many parameters and make exact and even approximate inference computationally intractable. Auspiciously, *sparsity* is also a desirable property for *parallelism*. Many algorithms that operate on graphical models factor with respect to the Markov Blanket of each variable. The **Markov Blanket** (see Figure 2.1d) of a variable is the set of neighboring variables in the MRF. Therefore sparse models, provide strong **locality** in computation which is a very desirable property in parallel algorithms. In fact the new parallel abstraction we develop in Chapter 5 is built around the locality derived from sparsity in the MRF and the idea that computation factorizes over Markov blankets.

Factor Graphs

In some settings it may be more convenient to reason about the **factor graph** corresponding to the factorized distribution. A factor graph is an undirected bipartite graph where the vertices $V = \mathbf{X} \cup \{f_\mathbf{A}\}_{\mathbf{A}\in\mathcal{F}}$ correspond to all the variables $\mathbf{X}$ on one side and all the factors $\{f_\mathbf{A}\}_{\mathbf{A}\in\mathcal{F}}$ on the other side. Undirected edges $E = \{\{f_\mathbf{A}, X_i\} : i \in \mathbf{A}, \mathbf{A} \in \mathcal{F}\}$ connect factors with the variables in their domain. To simplify notation, we will use $f_i \in V$ and $X_j \in V$ to refer to vertices in the factor when we wish to distinguish between factors and variables, and $i, j \in V$ otherwise. As with the MRF we define $\mathcal{N}_i \subseteq V$ as the neighbors of $i \in V$ in the factor graph. In Figure 2.1e we illustrate the factor graph for the image denoising problem.

2.1.3 Inference

There are three primary operations within graphical models: inference, parameter learning, and structure learning. Given the structure and parameters of the model, **inference** is the process of computing or estimating marginals:

$$\mathbf{P}\left(\mathbf{X}_\mathcal{S} \,|\, \theta\right) = \sum_{\mathbf{x}_{\{1,\ldots,n\}\setminus\mathcal{S}}} \mathbf{P}\left(\mathbf{X}_\mathcal{S}, \mathbf{x}_{\{1,\ldots,n\}\setminus\mathcal{S}} \,|\, \theta\right) \tag{2.14}$$

Estimating marginal distributions is an essential step in probabilistic reasoning and is needed to construct conditionals and posteriors, make predictions, and evaluate the partition function $Z(\theta)$. For example, by constructing the marginals of a factor graph representing the interactions between protein side chains we can compute the likely orientations of each side chain and bound the energy of a particular configuration. Alternatively, the marginals can be used in various learning algorithms to optimize model parameters.

Unfortunately, exact inference is NP-hard in general [Cooper, 1990] and even computing bounded approximations is known to be NP-hard [Roth, 1993]. Since exact inference is not tractable in most large-scale real-world graphical models we typically resort to approximate inference algorithms. Fortunately, there are many of approximate inference algorithms which typically perform well in practice. In addition we can often use the approximations made by approximate inference algorithms to introduce additional parallelism. In some cases (e.g., Section 3.3) we can introduce new approximations that are tailored to the objective of increasing parallelism with a bounded impact on the resulting error in inference.

2.1.4 Parameter Learning

Parameter learning in graphical models is used to choose the value of the parameters θ, usually by fitting the model to data. The parameters may correspond to individual entries in a probability table or the weights in functions (factors) over sets of variables. Often many factors will depend on a single parameter (shared parameter). In the image denoising task their are only two parameters which are shared across all the factors and act as weights.

Parameter learning is typically accomplished by maximizing the log-likelihood of the data:

$$\begin{aligned} \theta^{\text{MLE}} &= \arg\max_{\theta} \sum_{i=1}^{d} \log \mathbf{P}\left(\mathbf{x}_{\mathbf{A}}^{(i)} \,|\, \theta\right) && (2.15) \\ &= \arg\max_{\theta} \sum_{i=1}^{d} \sum_{\mathbf{A}\in\mathcal{F}} \log f_{\mathbf{A}}(\mathbf{x}_{\mathbf{A}}^{(i)} \,|\, \theta) - d \log \sum_{\mathbf{x}} \prod_{\mathbf{A}\in\mathcal{F}} f_{\mathbf{A}}(\mathbf{x}_{\mathbf{A}} \,|\, \theta) && (2.16) \end{aligned}$$

Unfortunately, while the left term (unnormalized log-likelihood) in Eq. (2.16) is tractable to compute, the right term (log-partition function) is intractable and must be computed through approximate inference. Therefore by focusing on improving the scalability of inference we can also improve the scalability of parameter learning.

In many applications parameter learning is not necessary. In a purely Bayesian formulation, priors are placed on parameters and the actual parameter values are treated as variables which are eliminated through marginalization during inference. Often domain knowledge, like the physical energy between interacting molecules, is used to select parameter values.

2.1.5 Structure Learning

The final and typically most challenging task in graphical models, is structure learning. Here the structure of the MRF, or the model factorization encoded by $\mathcal{F}$, is learned from data. Structure learning is most challenging because it adds a combinatorial search on-top of the already intractable problems of learning and inference. Moreover, in order to apply structure learning we need a method to impose factors over the maximal cliques in the MRF and then the ability to learn optimal values for the factor parameters θ given the structure. Therefore, structure learning requires parameter learning and consequently inference as nested subroutines.

In many real-world applications of graphical models, the structure is typically predetermined from prior knowledge. For example, in many vision applications the structure is a regular grid or mesh connecting nearby pixels and objects. Alternatively, in language modeling applications the structure is determined by the presence of words in documents. Even in our social network example, the graphical model structure is derived from the friend network. Therefore, in this thesis we will assume that the structure and parameters of the model are given, however we believe that the search for *scalable parameters and substructures* is an exciting direction for future research.

2.1.6 Summary of Important Graphical Models Properties in the Parallel Setting

Graphical models provide a standardized representation of complex probabilistic relationships enabling the design and implementation of general purpose probabilistic reasoning algorithms. Moreover, by providing a small suite of efficient parallel algorithms we can address a wide range of applications. In this thesis we focus on inference algorithms which are the key computational bottleneck in both learning an applying graphical models.

The same sparse graph structure which enables graphical models to efficiently encode large probability distributions also exposes the fundamental dependencies in computation and present an opportunity to leverage the wealth of existing work in parallel graph algorithms. For example our work in Gibbs sampling demonstrates that we can exploit the factorization of single variable Gibbs updates along with classical ideas from job scheduling on a dependency graph to construct a parallel Gibbs sampler that is provably correct and has theoretically quantifiable mixing acceleration.

The inherent complexity of learning and inference in large graphical models often necessitates approximations. By restructuring or simply reinterpreting the approximations made by efficient sequential algorithms we have considerable latitude to introduce additional parallelism. For example, in Chapter 3 we show how approximations can introduce substantial parallel gains.

To conclude this section we briefly summarize several of the key properties of probabilistic graphical models that will enable us to achieve the proposed objectives:

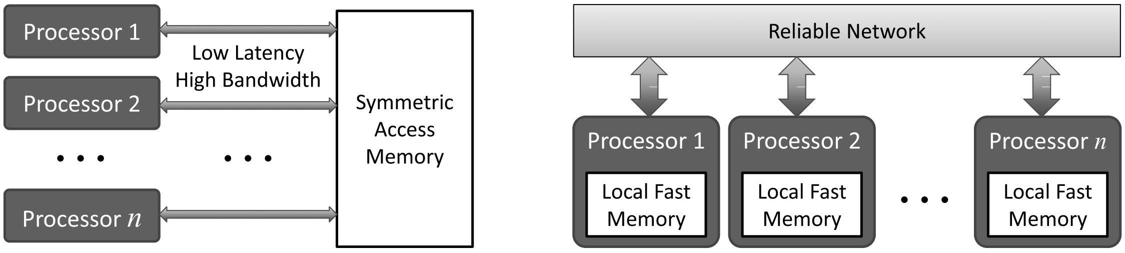
- Graphical models provide a *standardized representation* enabling the design and implementation of a small core set of parallel algorithms which span a wide range of real-world applications.
- Efficient graphical models have a *sparse dependency structure* which facilitates efficient factorized (local) parallel computation.
- The *explicit graph structure* of graphical models enables us to leverage classic work in parallel graph algorithms and job scheduling.
- The inherent complexity of learning and inference in large graphical models often necessitates *approximations* which can be used to expose substantial additional parallelism.

2.2 Parallel and Distributed Systems

Unlike the sequential setting where most processor designs faithfully execute a single random access computational model, the parallel setting has produced a wide variety of parallel architectures. In this work we focus on Multiple Instruction Multiple Data (MIMD) parallel architectures which includes multicore processors and distributed clusters. The MIMD architecture permits each processor to asynchronously execute its own sequence of instruction (MI) and operate on separate data (MD). Furthermore we divide the MIMD architectures into Shared Memory (SM-MIMD) represented by multi-core processors and Distributed Memory (DM-MIMD) represented by large commodity clusters and cloud systems. This section will briefly review the design and unique opportunities and challenges afforded by shared and distributed memory MIMD systems in the context of parallel machine learning.

2.2.1 Shared Memory

The shared-memory architectures are a natural generalization of the standard sequential processor to the parallel setting. In this work we assume that each processor is identical and has symmetric access to a single shared memory. All inter-processor communication is accomplished through shared memory and



(a) Shared Memory

(b) Distributed Memory

Figure 2.2: The two common parallel architectures encountered in typical commodity hardware. **(a)** The standard shared memory architectural model consists of a collection of processors that have low latency high bandwidth access to a single uniform address space. All inter-processor communication is accomplished through the shared memory. **(b)** The standard distributed memory architectural model consists of a collection of processors each with their own local memory which are able to communicate by passing messages across a reliable but typically slow communication network.

is typically very fast (on the order of a memory read). Consequently, shared memory architectures are particularly convenient because they do not require partitioning the algorithm state and enable frequent interaction between processors.

The class of parallel architectures that implement symmetric access to shared memory are commonly referred to as symmetric multi-processors (SMP). Most modern SMP systems are composed of one or more multi-core processors. Each multi-core chip is itself composed of several processors typically sharing a local cache. These systems must therefore implement sophisticated mechanisms to ensure cache coherency, limiting the potential for parallel scaling. Nonetheless, moderately sized mutli-core/SMP can be found in most modern computer systems and even in many embedded systems. It is expected that the number of cores in standard SMP systems will continue to increase [Asanovic et al., 2006].

While multi-core systems offer the convenience of shared memory parallelism they introduce challenges in data intensive computing. By dividing the already limited bandwidth to main memory over several processors, multi-core systems quickly become memory bandwidth constrained. Therefore, efficient multi-core algorithms need to carefully manage memory access patterns.

2.2.2 Distributed Memory and Cluster Computing

Distributed memory systems are found in systems ranging from small clusters to elastic cloud computing services like EC2. Distributed memory systems are composed of a collection of independent processors each with its own local memory and connected by a relatively slow communication network. Distributed memory algorithms must therefore address the additional costs associated with communication and distributed coordination. While there are many performance and connectivity models for distributed computing [Bertsekas and Tsitsiklis, 1989], most provide point-to-point routing, a maximum latency which scales with the number of processors, and a limited bandwidth.

Unlike the shared memory setting, the distributed setting often considers systems with network and processor failures. As clusters become larger, rare failure modes become more common and the chances of machine and network failure increases. The ability to recover from and in many cases tolerate limited system failure has been a focus of recent work in distributed computing systems and cloud environments. For many of the moderately sized clusters used in research some of the emphasis on fault-tolerance may be unnecessary (see Section 5.7.3). In our work on graphical model inference algorithms we will initially assume that all resources remain available throughout execution and that all messages eventually reach their destination. We then relax this assumption in the GraphLab system by introducing relatively simple check-point and recover fault-tolerance model.

Typical distributed memory systems range from hundreds of processors to thousands of processors. These system employ fast commodity networks with switch hierarchies or highly tuned system specific interconnects. Often distributed memory systems are composed of smaller shared memory systems, typically having several multi-core processors at each node. Here we will use the term node to refer to a single machine on the network and we will use the term processor to refer each processing element. For instance, a network of 8 machines, each with 2 quad-core CPUs, has 8 nodes, 64 processors.

By eliminating the need for cache coherency, the distributed memory model permits the construction of substantially larger systems and also offers several key advantages to highly data intensive algorithms. Because each processor has its own memory and dedicated bus, the distributed memory setting provides linear scaling in memory capacity and memory bandwidth. In heavily data intensive algorithms linear scaling in memory can provide substantial performance gains by eliminating memory bottlenecks.

Because of the latency associated with network communication and the limited network bandwidth, efficient algorithms in the distributed memory model must minimize inter-processor communication. Consequently, efficient distributed memory algorithms must deal with the challenges associated with partitioning both the state and execution in a way that minimizes network congestion while still ensuring that no one processor has disproportionally greater work.

Chapter 3

Probabilistic Inference: Belief Propagation

In this chapter we evaluate the thesis statement in the context of the widely used [Baron et al., 2010, Jaimovich et al., 2006, Lan et al., 2006, Singla and Domingos, 2008, Sun et al., 2002] Loopy Belief Propagation (BP) approximate inference algorithm. Many [Kang et al., 2010, Mendiburu et al., 2007, Sun et al., 2002] have proposed synchronous parallelizations of the Loopy BP algorithm. We will show that synchronous parallelization of Loopy BP is highly inefficient and propose a series of more efficient parallelizations by adhering to the design methodology described within the thesis statement. In the process we develop the SplashBP algorithm which combines new scheduling ideas to address the limitations of existing *sequential* BP algorithms and achieves theoretically optimal performance. Finally, we present how to efficiently implement BP algorithms in the distributed setting by addressing the challenges of distributed state and load-balancing. We provide both theoretical and real-world experimental analysis along with implementation specific details addressing locking and efficient data-structures. In addition, C++ code for the algorithms and experiments presented in this chapter can be obtained from our online repository at `http://gonzalezlabs/thesis`.

3.1 Belief Propagation

A core operation in probabilistic reasoning is **inference** (see Section 2.1.3) — the process of computing the probability of an event. While inference is NP-hard [Cooper, 1990] in general, there are several popular approximate inference algorithms which typically perform well in practice. Unfortunately, approximate inference is still computationally intensive [Roth, 1993] and therefore can benefit from parallelization. In this chapter we parallelize **Loopy Belief Propagation** (or BP in short) which is used in a wide range of ML applications [Baron et al., 2010, Jaimovich et al., 2006, Lan et al., 2006, Singla and Domingos, 2008, Sun et al., 2002]. For a more detailed review of the BP and it generalizations we recommend reading the tutorial by Yedidia et al. [2003].

Belief propagation (BP), or the Sum-Product algorithm, was introduced by Pearl [1988] as an *exact* inference procedure for tree-structured models but was later popularized as Loopy Belief Propagation, an *approximate* inference algorithm for models with cycles or "loops." However, for simplicity we will refer to Loopy Belief Propagation as BP.

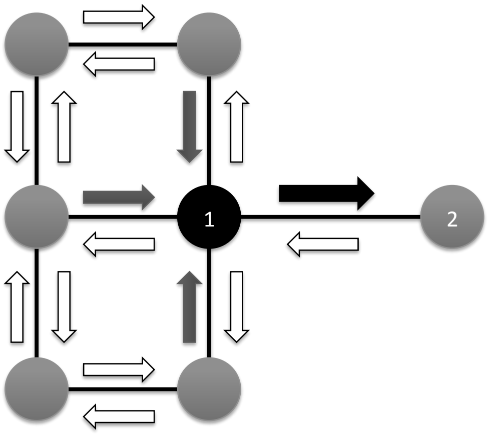


Figure 3.1: The solid message leaving the solid center vertex depends only on the shaded messages entering the center vertex.

Computation in the BP algorithm *factorizes* over the graphical model structure. The BP algorithm associates a vector of parameters called a **message**[1] along each direction of every edge in the factor graph. The messages $m_{X_i \to f_{\mathbf{A}}}$ and $m_{f_{\mathbf{A}} \to X_i}$ along the edge connecting variable X_i to factor $f_{\mathbf{A}}$ are positive functions (tables) mapping assignments to X_i to the positive reals. In loopy graphs, it is difficult to provide a direct probabilistic interpretation of the messages. While the messages technically correspond to Langrange multipliers in the Bethe approximation, it may help to think of the message $m_{f_{\mathbf{A}} \to X_i}$ as encoding a distribution over the variable X_i.

Starting with messages encoding uniform distributions, the BP algorithm iteratively recomputes (updates) messages using the following update (fixed-point) equations:

$$m_{X_i \to f_{\mathbf{A}}}(x_i) \quad \propto \quad \prod_{f_{\mathbf{B}} \in \mathcal{N}_{X_i} \setminus f_{\mathbf{A}}} m_{f_{\mathbf{B}} \to X_i}(x_i) \tag{3.1}$$

$$m_{f_{\mathbf{A}} \to X_i}(x_i) \quad \propto \quad \sum_{\mathbf{x_A} \setminus x_i} f_{\mathbf{A}}(\mathbf{x_A}) \prod_{X_k \in \mathcal{N}_{f_{\mathbf{A}}} \setminus X_i} m_{X_k \to f_{\mathbf{A}}}(x_k) \tag{3.2}$$

where Eq. (3.1) is the message sent from variable X_i to factor $f_{\mathbf{A}}$ and Eq. (3.2) is the message sent from factor $f_{\mathbf{A}}$ to variable X_i. The sum, $\sum_{\mathbf{x_A} \setminus x_i}$, is computed over all assignments to $\mathbf{x_A}$ excluding the variable x_i, and the product, $\prod_{X_k \in \mathcal{N}_{f_{\mathbf{A}}} \setminus X_i}$, is computed over all neighbors of the vertex $f_{\mathbf{A}}$ excluding vertex X_i. To ensure numerical stability the messages are typically normalized.

A critical property of the BP update equations (Eq. (3.1) and Eq. (3.2)) is that they only depend on the inbound messages associated with edges adjacent to the corresponding vertex (see Figure 3.1). As a consequence the BP update equations demonstrate the *factorized* locality structure needed for efficient parallel and distributed computation. In addition we will also exploit the property that the outbound message does *not* depend on the inbound message along the same edge.

The update equations are iteratively applied until a convergence condition is met. Typically, convergence is

[1] Even though "message" is the traditional terminology and the words "send" and "receive" are frequently used, it should not be interpreted as a form of communication. The "message" is simply a numeric vector used in the derivation of the loopy BP algorithm in Pearl [1988].

defined in terms of a bound on the change in message values between iterations:

$$\max_{(i,j)\in E} \left|\left| m_{i\to j}^{(\text{new})} - m_{i\to j}^{(\text{old})} \right|\right|_1 \leq \beta. \tag{3.3}$$

for a small constant $\beta \geq 0$. Unfortunately, in cyclic graphical models there are few convergence guarantees [Ihler et al., 2005, Mooij and Kappen, 2007, Tatikonda and Jordan, 2002].

At convergence, the variable and clique marginals, also called the beliefs (b) are estimated using:

$$\begin{aligned} \mathbf{P}\left(X_i = x_i\right) &\approx b_{X_i}(x_i) \propto \prod_{f_{\mathbf{A}} \in \mathcal{N}_{X_i}} m_{f_{\mathbf{A}} \to X_i}(x_i) && (3.4) \\ \mathbf{P}\left(\mathbf{X_A} = \mathbf{x_A}\right) &\approx b_{\mathbf{X_A}}(\mathbf{x_A}) \propto f_{\mathbf{A}}(\mathbf{x_A}) \prod_{X_j \in \mathcal{N}_{f_{\mathbf{A}}}} m_{X_j \to f_{\mathbf{A}}}(x_j). && (3.5) \end{aligned}$$

In other words, the approximate marginal distribution of a variable is simply the (normalized) product of all of its incoming messages. Similarly, the approximate marginal of all variables within a factor can be estimated by multiplying the factor with all of its incoming messages.

3.1.1 The Message Schedule

The order in which messages are updated is called the **schedule** and plays a central role in efficient BP. For instance, in tree graphical models, a simple procedure known as the forward-backward schedule was shown by Pearl [1988] to yield exact marginals using $O(2\,|E|)$ message calculations. First, messages are computed starting from the leaves and in the direction of an arbitrarily chosen root. The process is then reversed computing all the messages in the opposite direction. The reader may observe from Eq. (3.1) and Eq. (3.2) that this forward-backward (or perhaps upward-downward) schedule achieves exact convergence. That is, re-evaluation of any message using the message update equations, will not change the message values.

Unfortunately, choosing the best schedule on loopy graphs is often difficult and can depend heavily on the factor graph structure and even the model parameters. For simplicity, many applications of loopy BP adopt a **synchronous** schedule in which all messages are *simultaneously* updated using messages from the previous iteration. Alternatively, in some cases an **asynchronous** schedule is employed, in which messages are updated *sequentially* using the most recent inbound messages. For example, the popular **round-robin** asynchronous schedule, sequentially updates the messages in fixed order which is typically a random permutation over the vertices. Advances by Elidan et al. [2006] and Ranganathan et al. [2007] have focused on **dynamic** asynchronous schedules, in which the message update order is determined as the algorithm proceeds. Other work by Wainwright et al. [2001] focus on tree structured schedules, in which messages are updated along collections of spanning trees.

In this chapter we demonstrate that by varying the BP schedule we can affect the *efficiency*, *speed*, and *parallelism* of BP. By applying the GrAD design methodology we develop computationally efficient parallel schedules that simultaneously focus computation and respect the sequential structure of the underlying graphical model while exposing a high-degree of parallelism. In the process we unify and refine the work on dynamic scheduling [Elidan et al., 2006] and tree scheduling [Wainwright et al., 2001] to achieve state-of-the-art performance in the sequential, parallel, and distributed settings.

Algorithm 3.1: Synchronous BP

```
Input: Graph $G = (V, E)$ and all messages $\{m_{i \to j} : \forall (i, j) \in E\}$
while not converged do
    // Embarrassingly Parallel Computation Step
    forall $j \in \mathcal{N}_v$ in the neighbors of $v$ do in parallel
        Compute Message $m_{v \to j}^{(\text{new})}$ using $\left\{ m_{i \to v}^{(\text{old})} \right\}_{i \in \mathcal{N}_v}$
    // Synchronization Phase
    Swap($m^{(\text{old})}$, $m^{(\text{new})}$)
```

3.1.2 Opportunities for Parallelism in Belief Propagation

Because BP update equation *factor* the inference computation over the graph they exposes substantial **graph-level** parallelism. Moreover, all the messages in the graph could be computed simultaneously and in parallel: leading to the Synchronous BP algorithm which we will present next in Section 3.2. In addition BP exposes opportunities for more fine-grained **factor-level** parallelism through the individual message calculations (sums and products) which can be expressed as matrix operations which can be parallelized relatively easily [Bertsekas and Tsitsiklis, 1989]. For typical message sizes where the number of assignments is much less than the number of vertices ($\mathcal{X}_i << n$), graph level parallelism provides more potential gains than factor level parallelism. For instance, the message update equations do not offer much parallelism if all variables are binary. Therefore, we will ignore factor level parallelism in this chapter and instead focus on graph level parallelism. Running time will be measured in terms of the number of message computations, treating individual message updates as atomic unit time operations.

Pennock [1998] and Xia et al. [2009] explore parallel transformations of the graph which go beyond the standard message calculations. While these techniques expose additional parallelism they require low tree-width models and tractable exact inference, which are unreasonably strong requirements when operating on massive real-world distributions. In acyclic models these techniques can achieve running times that are logarithmic in the tree height (as apposed to the linear running times presented in this chapter) but require considerable fine grained parallelism. In this chapter we restrict our attention to message passing algorithms on the original graph and do not require low tree-width models.

3.2 Synchronous BP

Running BP with a synchronous message schedule leads to the highly parallel Synchronous BP algorithm shown in Alg. 3.1. In fact Synchronous BP perfectly fits the classic Bulk Synchronous Parallel (BSP) model of computation (described in greater detail in Section 5.3). The BSP model divides computation into *super-steps* each consisting of two phases. During the first phase, each processor executes independently updating its local BP messages. In the second phase, all processors exchange BP messages. Finally, a barrier synchronization is performed before the next iteration, ensuring all processors compute and communicate in lockstep.

In the shared memory setting messages are exchanged by swapping between old and new message buffers at the end of each iteration. In the distributed memory setting messages must be sent over the communication

Algorithm 3.2: Map Function for Synchronous BP

```
Input: A vertex $v \in V$ and all inbound messages $\{m_{i \to v}\}_{i \in \mathcal{N}_v}$
Output: Set of outbound messages as key-value pairs $(j, m_{v \to j})$
forall $j \in \mathcal{N}_v$ in the neighbors of $v$ do in parallel
    Compute Message $m_{v \to j}$ using $\{m_{i \to v}\}_{i \in \mathcal{N}_v}$
    return key-value pair $(j, m_{v \to j})$
```

Algorithm 3.3: Reduce Function for Synchronous BP

```
Input: The key-value pairs $\{(v, m_{i \to v})\}_{i \in \mathcal{N}_v}$
Output: The belief $b_v$ for vertex $v$ as well as the $\{(v, m_{i \to v})\}_{i \in \mathcal{N}_v}$ pairs $(j, m_{v \to j})$
Compute the belief $b_v$ for $v$ using $\{(v, m_{i \to v})\}_{i \in \mathcal{N}_v}$
Return $b_v$ and $\{(v, m_{i \to v})\}_{i \in \mathcal{N}_v}$
```

network. In either case Synchronous BP requires two copies of each message to be maintained at all times. When the messages are updated their new the values in $m^{(\text{old})}$ are used as input, while the resultant values are stored in $m^{(\text{new})}$. In addition, the task of assessing convergence is relatively straight forward for Synchronous BP. After each iteration all processors (or machines in the distributed setting) vote to halt based on whether all the messages they have compute satisfy the termination condition (Eq. (3.3)).

3.2.1 Synchronous BP in MapReduce

The Synchronous BP algorithm may also be expressed in the context of the popular Map-Reduce framework. The Map-Reduce framework, introduced by Dean and Ghemawat [2004], simplifies designing and and implementation large scale parallel algorithms and has therefore been widely adopted by the data-mining and machine learning community [Chu et al., 2006]. Synchronous BP can naturally be expressed as an iterative MapReduce algorithm where the `Map` operation (defined in Alg. 3.2) is applied to all vertices and emits destination-message key-value pairs and the `Reduce` operation (defined in Alg. 3.3) joins messages at their destination vertex, updates the local belief, and prepares for the next iteration. It is important to note that the MapReduce abstraction was not originally designed for iterative algorithms, like belief propagation, and therefore standard implementations, like Hadoop, incur a costly communication and disk access penalty between iterations.

3.2.2 Synchronous BP Runtime Analysis

While it is not possible to analyze the running time of Synchronous BP on a general cyclic graphical model, we can analyze the running time in the context of tree graphical models. In Theorem 3.2.1 we characterize the running time of Synchronous BP when computing *exact* marginals (with $\beta = 0$).

Theorem 3.2.1 (Exact Synchronous BP Running Time)**.** *Given an acyclic factor graph with n vertices, longest path length l, and $p \leq 2(n-1)$ processors, parallel synchronous belief propagation will compute exact marginals in time (as measured in number of vertex updates):*

$$\Theta\left(\frac{nl}{p} + l\right).$$

Proof of Theorem 3.2.1. The proof follows naturally from the optimality of the standard forward-backward scheduling on tree-structured models. On the k^{th} iteration of Synchronous BP, all messages have traveled a distance of k in all directions. We know from the forward-backward algorithm that the furthest distance we must push messages is l. Therefore it will take at most l iterations for Synchronous BP to obtain *exact* marginals.

Each iteration requires $2(n-1)$ message calculations, which we divide evenly over p processors. Therefore it takes $\lceil 2\left(n-1\right)/p \rceil$ time to complete a single iteration. Thus the total running time is:

$$l\left\lceil \frac{2\left(n-1\right)}{p} \right\rceil \leq l\left(\frac{2\left(n-1\right)}{p}+1\right) \in \Theta\left(\frac{nl}{p}+l\right)$$

□

If we consider the running time given by Theorem 3.2.1 we see that the n/p term corresponds to the parallelization of each synchronous update. The length l of the longest path corresponds to the limiting sequential component which cannot be eliminated by scaling the number of processors. As long as the number of vertices is much greater than the number of processors, the Synchronous BP algorithm achieves nearly linear parallel scaling and therefore appears to be an optimal parallel algorithm. However, an optimal parallel algorithm should also be *efficient*. That is, the total work done by all processors should be asymptotically equivalent to the work done by a single processor running the *optimal* sequential algorithm. By examining the simple chain graphical model we will show that there are $O\left(n\right)$ unnecessary parallel message computations at each iteration leading to a highly suboptimal parallel algorithm.

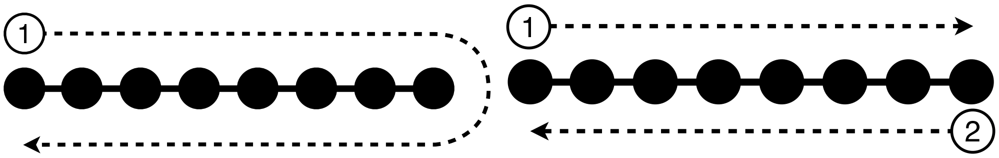
3.2.3 Analysis of Synchronous Belief Propagation on Chains

To illustrate the inefficiency of Synchronous BP, we analyze the running time on a chain graphical model with n vertices. Chain graphical models act as a theoretical benchmark by directly capturing the limiting sequential structure of message passing algorithms and can be seen as a sub-problem in both acyclic and cyclic graphical models. Chain graphical models also represent the worst case acyclic graphical model structure for parallel belief propagation because they contain only a single long path and do not expose any opportunities[2] for parallelism. Conversely, the branching structure of trees permits *multiple* longest paths and exposes an opportunity for additional parallelism by running each path separately. Even in cyclic graphical models, the longest paths in the unrolled computation graph again reduces to a chain graphical model.

It is well known that the forward-backward schedule (Figure 3.2a) for belief propagation on chain graphical models is optimal. The forward-backward schedule, as the name implies, sequentially computes messages from $m_{1\to 2}$ to $m_{n-1\to n}$ in the forward direction and then sequentially computes messages from $m_{n\to n-1}$ to $m_{2\to 1}$ in the backward direction. The running time of this simple schedule is therefore $\Theta\left(n\right)$ or exactly $2(n-1)$ message calculations.

If we run the Synchronous BP algorithm using $p = 2(n-1)$ processors on a chain graphical model of length n, we obtain a running time of exactly $n-1$. This means that parallel Synchronous BP algorithm obtains only a *factor of two* speedup using two processors per edge, almost twice as many processors as the number of vertices. More surprisingly, if we use fewer than $n-1$ processors, the parallel synchronous

[2] While it is true that rake-compress style algorithms as described by Miller and Reif [1985] could be applied to chain graphical models they do not generalize to cyclic models.



(a) Single Sequential

(b) Optimal Parallel

Figure 3.2: **(a)** The optimal forward-backwards message ordering for exact inference on a chain using a single processor. **(b)** The optimal message ordering for exact inference on a chain using two processors.

algorithm will be *slower* than the simple sequential forward-backward algorithm running on a single processor. Finally, if we use any constant number of processors (for example $p = 2$), then the parallel Synchronous BP algorithm will run in *quadratic* time while the sequential single processor algorithm will run in *linear* time.

What is the optimal *parallel* schedule for belief propagation on chain graphical models? Recall that in the message update equations (Eq. (3.1) and Eq. (3.2)), the message $m_{i \to j}$ does not depend on the message $m_{j \to i}$ (see Figure 3.1). In the sequential forward-backward algorithm, messages in the forward sweep do not interact with messages in the backward sweep and therefore we can compute the forward and backward sweep *in parallel* as shown in Figure 3.2b. Using only $p = 2$ processors, simultaneously computing messages *sequentially* in the forward and backward directions as shown in Figure 3.2b, we obtain a running time of $n - 1$ and achieve a *factor of two* speedup.

While messages may be computed in any order, information is propagated *sequentially*. On every iteration of Synchronous BP only a few message computations (in the case of chain graphical models only two message computations) contribute to convergence while the rest are *wasted*. Unfortunately, there are no parallel schedules which achieve greater than a factor of two speedup for *exact* inference on arbitrary chain graphical models. Fortunately, there are parallel schedules which can achieve substantially better scaling by exploiting a frequently used *approximation* in loopy BP.

3.3 The τ_ϵ Approximation

In almost all applications of loopy BP, the convergence threshold β in Eq. (3.3) is set to a small value greater than zero and the algorithm is terminated prior to reaching the true fixed-point. Even when $\beta = 0$, the fixed floating point precision of discrete processors result in early termination. Because the resulting approximation plays an important role in studying parallel belief propagation, we provide a brief theoretical characterization.

We can represent a single iteration of synchronous belief propagation by a function U_{BP} which maps all the messages $m^{(t)}$ on the t^{th} iteration to all the messages $m^{(t+1)} = U_{\text{BP}}(m^{(t)})$ on the $(t+1)^{\text{th}}$ iteration. The fixed point is then the set of messages $m^* = U_{\text{BP}}(m^*)$ that are invariant under U_{BP}. In addition, we define a max-norm for the message space

$$\left|\left|m^{(t)} - m^{(t+1)}\right|\right|_\infty = \max_{(i,j) \in E} \left|\left|m^{(t)}_{i \to j} - m^{(t+1)}_{i \to j}\right|\right|_1, \tag{3.6}$$

which matches the norm used in the standard termination condition, Eq. (3.3).

Definition 3.3.1 (Definition of τ_ϵ-approximation). *We define τ_ϵ as:*

$$\tau_\epsilon = \min_t t \quad s.t. \quad \left|\left| m^{(t)} - m^* \right|\right|_\infty \leq \epsilon, \tag{3.7}$$

*the number of synchronous iterations required to be within an ϵ ball of the BP fixed-point. Therefore a τ_ϵ-**Approximation** is the approximation obtained from running synchronous belief propagation for τ_ϵ iterations.*

A common method for analyzing fixed point iterations is to show (assume) that the U_{BP} is a contraction mapping and then use the contraction rate to bound the number of iterations for an ϵ level approximation of m^*. If U_{BP} is a max-norm contraction mapping then for the fixed point m^* and $0 \leq \alpha < 1$,

$$||U_{\text{BP}}(m) - m^*||_\infty \leq \alpha \, ||m - m^*||_\infty \, .$$

Work by Mooij and Kappen [2007] provide sufficient conditions for $U_{\text{BP}}(m)$ to be a contraction mapping under a variety of norms including the max-norm and the L_1-norm and shows that BP on acyclic graphs is guaranteed to be a contraction mapping under a particular spectral norm.

If the contraction rate α is known, and we desire an ϵ approximation of the fixed point, τ_ϵ is the smallest value such that $\alpha^{\tau_\epsilon} \, ||m_0 - m^*||_\infty \leq \epsilon$. This is satisfied by setting

$$\tau_\epsilon \leq \left\lceil \frac{\log(2/\epsilon)}{\log(1/\alpha)} \right\rceil . \tag{3.8}$$

Finally, in Eq. (3.9) we observe that the convergence criterion, $||m - f(m)||_\infty$ defined in Eq. (3.3), is a constant factor upper bound on the distance between m and the fixed point m^*. If we desire an ϵ approximation, it is sufficient to set the convergence criterion $\beta \leq \epsilon(1 - \alpha)$.

$$\begin{aligned}
||m - m^*||_\infty &= ||m - U_{\text{BP}}(m) + U_{\text{BP}}(m) - m^*||_\infty \\
&\leq ||m - U_{\text{BP}}(m)||_\infty + ||U_{\text{BP}}(m) - m^*||_\infty \\
&\leq ||m - U_{\text{BP}}(m)||_\infty + \alpha \, ||m - m^*||_\infty \\
||m - m^*||_\infty &\leq \frac{1}{1-\alpha} \, ||m - U_{\text{BP}}(m)||_\infty
\end{aligned} \tag{3.9}$$

In practice, the contraction rate α is likely to be unknown and U_{BP} will not be a contraction mapping on all graphical models. Furthermore, it may be difficult to determine τ_ϵ without first running the inference algorithm. Ultimately, our results *only* rely on τ_ϵ as a theoretical tool for comparing inference algorithms and understanding parallel convergence behavior. In practice, rather than constructing a β for a particular α and ϵ, we advocate the more pragmatic method of picking the smallest possible value that resources permit.

3.3.1 Graphical Interpretation

The τ_ϵ-Approximation also has a graphical interpretation. Intuitively, for a long chain graphical model with weak (nearly uniform) edge potentials, distant vertices are approximately independent. For a particular vertex, an accurate approximation to the belief may often be achieved by running belief propagation on a small subgraph around that vertex. More formally to achieve an ϵ level approximation for a particular message, we require a subgraph which contains all vertices that are reachable in τ_ϵ steps.

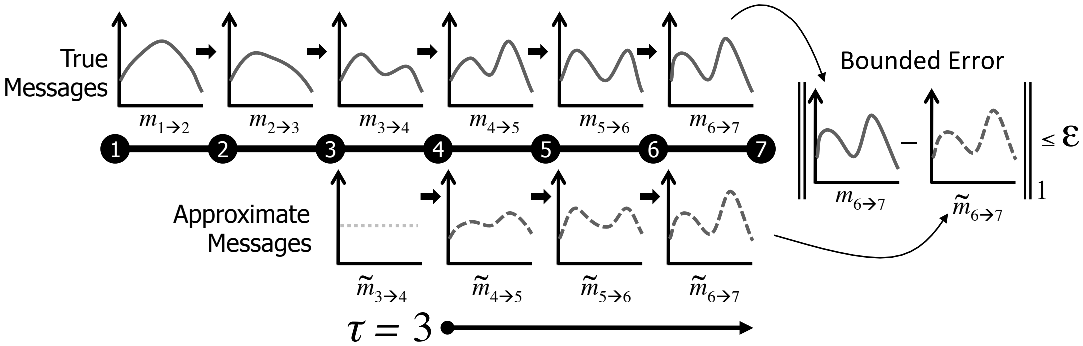


Figure 3.3: Above the chain graphical model we plot the sequence of messages from vertex 1 to vertex 7. For visual clarity these messages are plotted as continuous distributions. Below the chain we plot the approximate sequence of messages starting at vertex 4 assuming a uniform message is received from vertex 3. We see that the message that finally arrives at vertex 7 is only ϵ away from the original message. Therefore for an ϵ level approximation at vertex 7 messages must only be sent a distance of $\tau = 3$.

An illustration of the graphical interpretation is given in Figure 3.3. Suppose in the sequential forward pass we change the message $m_{3\to4}$ to a uniform distribution (or equivalently disconnect vertex 3 and 4) and then proceed to compute the rest of the messages from vertex 4 onwards. Assuming $\tau_\epsilon = 3$ for the chosen ϵ, the final approximate belief at b_7 would have less than ϵ error in some metric (here we will use L_1-norm) to the true probability $\mathbf{P}\left(X_{\tau_\epsilon+1}\right)$. Vertices τ_ϵ apart are therefore approximately independent. If τ_ϵ is small, this can dramatically *reduces* the length of the sequential component of the algorithm.

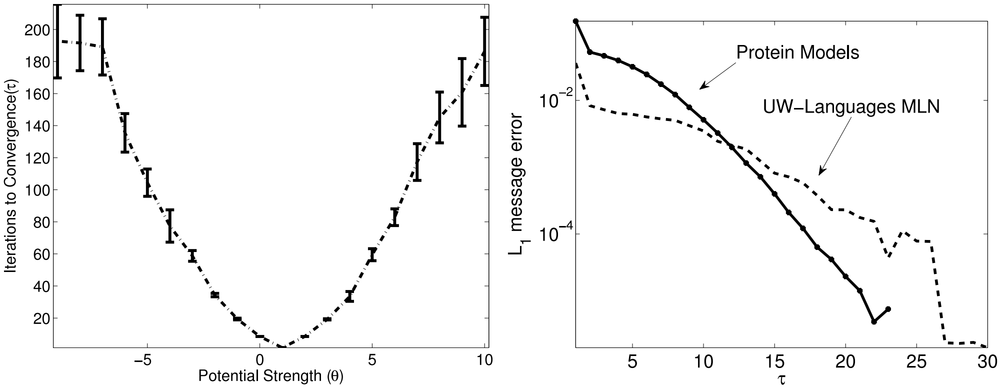
3.3.2 Empirical Analysis of τ_ϵ

The τ_ϵ structure of a graphical model is a measure of both the underlying chain structure as well as the *strength* of the *factors*. The strength of a factor refers to its coupling affect on the variables in its domain. A factor that assigns relatively similar probability to all configurations is much weaker than a factor that assigns disproportionately greater probability to a subset of its assignments. Graphical models with factors that tightly couple variables along chains will require a greater τ to achieve the same ϵ level approximation.

To illustrate the dependence τ_ϵ on the coupling strength of the factors we constructed a sequence of binary chain graphical models each with 1000 variables but with varying degrees of attractive and repulsive edge factors. We parameterized the edge factors by

$$f_{x_i,x_{i+1}} = \begin{cases} e^\theta & x_i = x_{i+1} \\ e^1 & \text{otherwise} \end{cases} \tag{3.10}$$

where θ is the “strength” parameter. When $\theta = 1$, every variable is independent and we expect $\tau_\epsilon = 1$ resulting in the shortest runtime. When $\theta < 1$ (corresponding to anti-ferromagnetic potentials) alternating assignments become more favorable. Conversely when $\theta > 1$ (corresponding to ferromagnetic potentials) matching assignments become more favorable. We constructed 16 chains at each potential with vertex potentials f_{x_i} chosen randomly from $\text{Unif}(0,1)$. We ran synchronous belief propagation until a fixed $\epsilon = 10^{-5}$ level approximation was achieved and plotted the number of iterations versus the potential



(a) τ_ϵ versus θ

(b) τ_ϵ versus θ

Figure 3.4: In **(a)** the number of iterations synchronous BP required to obtain a $\epsilon = 10^{-5}$ approximation on a 1000 variable chain graphical model is plotted against the strength of the pairwise potential θ. In **(b)** the log average message error is plotted as a function of the walk length on the protein pair-wise markov random fields and the UW-Languages MLN.

strength in Figure 3.4a. We observe that even with relatively strong potentials, the number of iterations, τ_ϵ, is still relatively small compared to the length of the chain.

We can further experimentally characterize the decay of message errors as a function of propagation distance. On several real networks we ran belief propagation to convergence (at $\beta = 10^{-10}$) and then simulated the effect of replacing an arbitrary message with a uniform distribution and then propagating that *error* along random paths in the graph. In Figure 3.4b we plot the message error as a function of distance averaged over 1000 random walks on various different factor graphs. Walks were terminated after the error fell below the original $\beta = 10^{-10}$ termination condition. In all cases we see that the affect of the original message error decays rapidly.

The concept of message decay in graphical models is not strictly novel. In the setting of online inference in Russell and Norvig [1995] used the notion of Fixed Lag Smoothing to eliminate the need to pass messages along the entire chain. Meanwhile, work by Ihler et al. [2005] has investigated message error decay along chains as it relates to robust inference in distributed settings, numerical precision, and general convergence guarantees for belief propagation in cyclic models. Our use of τ_ϵ is to theoretically characterize the additional parallelism exposed by an ϵ level approximation and to ultimate guide the design of the new optimal parallel belief propagation scheduling.

The τ_ϵ-Approximation enables us to achieve substantially greater parallelism in chain models with only a negligible loss in accuracy. The definition of τ_ϵ directly leads to an *improved* parallel running time for Synchronous BP:

Theorem 3.3.1 (τ_ϵ-Approximate Synchronous BP Running Time)**.** *Given an acyclic factor graph with n vertices and $p \leq 2(n-1)$ processors, parallel synchronous belief propagation will compute τ_ϵ-approximate messages in time (as measured in number of vertex updates):*

$$\Theta\left(\frac{n\tau_\epsilon}{p} + \tau_\epsilon\right).$$

Proof of Theorem 3.3.1. We know that each iteration will take $\lceil 2(n-1)/p \rceil$ time to complete. From

Definition 3.3.1 it follows that a τ_ϵ-approximation is obtained in exactly τ_ϵ iterations. □

The important consequence of Theorem 3.3.1 is that the fundamental sequential component of Synchronous BP depends on the *effective* chain length determined by τ_ϵ. Models with weaker variable interactions will have a smaller τ_ϵ and will permit greater parallelism. However, even with the reduced runtime afforded by the τ_ϵ-approximation, we can show that on a simple chain graphical model, the performance of Synchronous BP is still far from optimal when computing a τ_ϵ approximation. We formalize this intuition through the following lower bound on τ_ϵ-approximations for chain graphical models.

Theorem 3.3.2 (τ_ϵ-Approximate BP Lower Bound)**.** *For an arbitrary chain graph with n vertices and p processors, a lower bound for the running time of a τ_ϵ-approximation is:*

$$\Omega\left(\frac{n}{p} + \tau_\epsilon\right).$$

To prove Theorem 3.3.2 we introduce the concept of awareness. Intuitively, awareness captures the "flow" of message information along edges in the graph. If messages are passed *sequentially* along a chain of vertices starting at vertex i and terminating at vertex j then vertex j is aware of vertex i.

Definition 3.3.2 (**Awareness**)**.** *Vertex j is **aware** of vertex i if there exists a path $(v_1, \ldots, v_k)$ from $v_1 = i$ to $v_k = j$ in the factor graph G and a sequence of messages $[m^{(t_1)}_{v_1 \to v_2}, \ldots, m^{(t_k)}_{v_{k-1} \to v_k}]$ such that each message was computed after the previous message in the sequence $t_1 < \ldots < t_k$.*

Proof of Theorem 3.3.2. Because the messages sent in opposite directions are independent and the amount of work in each direction is symmetric, we can reduce the problem to computing a τ_ϵ-approximation in one direction (X_1 to X_n) using $p/2$ processors. Furthermore, to achieve a τ_ϵ-approximation, we need exactly $n - \tau_\epsilon$ vertices from $\{X_{\tau_\epsilon+1}, \ldots, X_n\}$ to be τ_ϵ left-aware (i.e., for all $i > \tau_\epsilon$, X_i is aware of $X_{i-\tau_\epsilon}$). By definition when $n - \tau_\epsilon$ vertices first become τ_ϵ left-aware the remaining (first) τ_ϵ vertices are maximally left-aware.

Let each processor compute a set of $k \geq \tau_\epsilon$ consecutive message updates in sequence (e.g., $[m^{(1)}_{1\to 2}, m^{(2)}_{2\to 3}, \ldots, m^{(k)}_{k-1\to k}]$). Notice that it is never beneficial to skip a message or compute messages out of order on a single processor since doing so cannot increase the number of vertices made left-aware. After the first τ_ϵ updates each additional message computation make exactly one more vertex left-aware. Therefore after k message computations each processor can make at most $k - \tau_\epsilon + 1$ vertices left-aware. Requiring all $p/2$ processors to act simultaneously, we observe that pre-emption will only decrease the number of vertices made τ_ϵ left-aware.

We then want to find a lower bound on k such that the number of vertices made left-aware after k iterations greater than the minimum number of vertices that must be made left-aware. Hence we obtain the following inequality:

$$\begin{aligned} n - \tau_\epsilon &\leq \frac{p}{2}\left(k - \tau_\epsilon + 1\right) \\ k &\geq \frac{2n}{p} + \tau_\epsilon\left(1 - \frac{2}{p}\right) - 1 \end{aligned} \tag{3.11}$$

relating required amount of work and the maximum amount of work done on the k^{th} iteration. For $p > 2$, Eq. (3.11) provides the desired asymptotic result. □

Algorithm 3.4: Atomic Parallel Round-Robin Algorithm

```
$\sigma \leftarrow$ Arbitrary permutation on $\{1, \dots, |V|\}$
Atomic integer $i \leftarrow 1$
do in parallel
    while Not Converged do
        // Reads the value of $i$ and increments $i$ atomically
        $j = AtomicFetchAndIncrement(i)$
        // Get the vertex to update
        $v = \sigma(j \mod |V|)$
        // Compute out messages after earlier neighbors complete.
        SendMessages($v$)
```

The bound provided in Theorem 3.3.2 is surprisingly revealing because it concisely isolates the fundamental parallel (n/p) and sequential (τ_ϵ) components of BP. Furthermore, as we scale models, holding their parameters fixed, τ_ϵ remains *constant* providing optimal parallel scaling albiet with τ_ϵ multiplicative inefficiency.

In the next few sections we will apply the GrAD methodology to introduce several more efficient parallel BP algorithms that will attempt to address this multiplicative dependence of $n\tau_\epsilon/p$ on τ_ϵ in Theorem 3.3.1 and improve upon memory efficiency and convergence.

3.4 Round Robin Belief Propagation

In contrast to the Synchronous BP algorithm in which updates all messages simultaneously, the more efficient forward-backward algorithm updates messages in a fixed *sequential* order. Once a message is updated its value is immediately visible to any future message calculations. More generally given any fixed ordering σ over the vertices, the inherently sequential Round-Robin BP algorithm iteratively **sweeps** the vertices in the order σ recomputing each of their out messages. As before, once a message is computed its new value is used in all message calculation for vertices later in σ.

We can obtain a parallel Round-Robin BP algorithm (Alg. 3.4) by choosing a particular sweep ordering σ and constructing a corresponding parallel execution. Rather than one processor iteratively updating messages in the order σ all p processors *asynchronously* update p vertices at a time. While each processor runs independently, the next vertex $v = \sigma(++j \mod |V|)$ is obtained by atomically advancing the shared schedule σ. After a processor gets the next vertex v it executes `SendMessages(`v`)` and checks for local convergence.

We introduce a locking scheme that prevents a processor from updating a vertex while a dependent message from an earlier vertex is being processed. The `SendMessages(`v`)` operation (Alg. 3.5) implements this locking scheme. When `SendMessages(`v`)` is invoked on vertex v it acquires a *write lock* on v and *read locks* on all adjacent vertices. As a consequence, concurrent invocations of `SendMessages` on adjacent vertices are serialized ensuring both that the *sequential* order is preserved and that only one processor is reading or writing the same a message value at the same time. To prevent deadlocks, all locks are acquired in the same order (σ) on all processors. Once all the adjacent locks have been acquired all out-bound messages are updated and then the locks are released.

Algorithm 3.5: SendMessages(v)

```
// Sort $v$ and its neighbors (done in advance)
$o \leftarrow$ Sort ($\mathcal{N}_v \cup \{v\}$) ordered by $\sigma$
// Lock $v$ and its neighbors in order
for $i \in [1, \ldots, |o|]$ do
    if $o_i = v$ then Acquire Write Lock on $v$
    else Acquire Read Lock on neighbor $o_i$
// Compute the new belief
$b_v \propto \prod_{u \in \mathcal{N}_v} m_{u \to v}$
if $v$ is a factor node then $b_v \propto f_v \times b_v$
// Compute outgoing messages
for $u \in \mathcal{N}_v$ do $m_{v \to u} \propto \sum_{\mathbf{x}_v \setminus \mathbf{x}_u} b_v / m_{u \to v}$
for $i \in [1, \ldots, |o|]$ do
    Release Lock on $o_i$
```

The locking mechanism used in SendMessages (Alg. 3.5) ensures that every sweep of the Round Robin BP algorithm is **serializable**: there is a corresponding sequential execution $\sigma^{(t)}$. It is possible for the serialization $\sigma^{(t)}$ to differ form σ the proposed permutation. However, for large cyclic models where σ is usually a random permutation, such variation is not typically not an issue.

In the shared memory setting the Round-Robin BP algorithm uses *atomic* operations (see Section 2.2.1) which are typically available on most modern processors and provide the coordination needed to maintain the shared schedule. In the distributed setting we further relax serializability constraints by partitioning (see Section 3.8.1) the Round-Robin schedule across machines allowing messages along the boundary between machines to be computed concurrently.

3.5 Wildfire Belief Propagation

While Round Robin BP tends to converge faster than Synchronous BP, the Round Robin BP algorithm can still lead to a significant amount of unnecessary computation. For example, consider a graphical model which comprises of two disconnected regions. The first region takes a Round Robin sweeps to converge, while the second region takes b sweeps, where $a << b$. Performing full Round Robin BP sweeps on the entire graph will therefore take b iterations to converge. The first region will run for $b - a$ more iterations than necessary resulting in substantial wasted computation.

A simple solution to the problem of over scheduling an already converged region of the model is to simply skip vertices that have converged. The resulting Wildfire algorithm shown in Alg. 3.6 is a direct parallelization of the algorithm proposed by [Ranganathan et al., 2007]. The Wildfire algorithm is almost identical to the Round Robin algorithm (Alg. 3.4) except that only messages that have changed significantly:

$$\left|\left| m_{i \to j}^{(\text{new})} - m_{i \to j}^{(\text{old})} \right|\right|_1 > \beta \tag{3.12}$$

are sent. To apply this definition we must compute all the new outbound messages every time a new

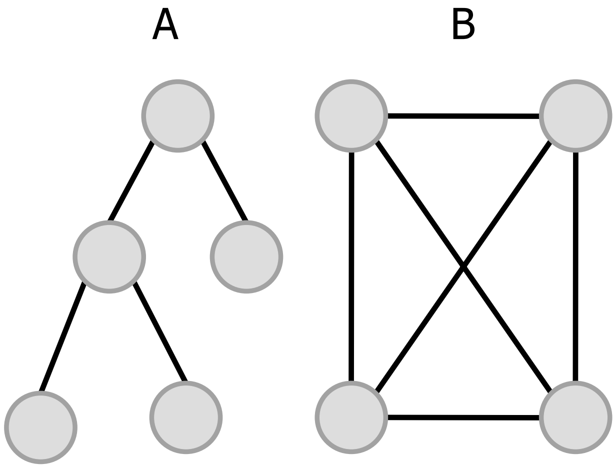


Figure 3.5: If Round Robin BP is used on this graph, the left part of the graph (A) will converge rapidly in a few iterations, while the right part of the graph (B) may take much longer to converge. Running full sweeps therefore wastes computation.

message arrives. However, only the outbound messages that have changed by more then β are sent and the rest are discarded. If a vertex has not received any messages since the last Round Robin sweep then it is not necessary to recompute its outbound messages.

3.6 Residual Belief Propagation

A natural extension of to the WildFire BP algorithm is to prioritize the reception of the most influential messages over messages that have not changed substantially. Elidan et al. [2006] introduced the Residual BP algorithm which prioritizes messages based on their **residual**:

$$r_{m_{i\to j}} = \left|\left| m_{i\to j}^{(\text{new})} - m_{i\to j}^{(\text{old})} \right|\right|_1 \tag{3.13}$$

which is analogous to the skipping condition used in the Wildfire belief propagation algorithm. The residual prioritization greedily updates the message that most violates the termination condition. In settings where belief propagation is a contraction mapping, Elidan et al. [2006] showed that the largest residual is an upper bound on the distance to fixed-point and therefore the residual heuristic is greedily minimizing and upper bound on the underlying BP energy.

Every time a vertex receives one message it actually must "re-receive" all messages into that vertex. While it is possible to "incrementally" receive each new message doing so would be wasteful since every received message would require completely recomputing all out-bound messages. Therefore prioritizing messages actually corresponds to prioritizing vertices. Using Eq. (3.13) corresponds to using the vertex priority:

$$r_j = \max_i \left|\left| m_{i\to j}^{(\text{new})} - m_{i\to j}^{(\text{old})} \right|\right|_1 \tag{3.14}$$

which prioritizes vertices based on the new message with largest change. Intuitively, vertex residuals capture the amount of new information available to a vertex. Recomputing outbound messages from a vertex with unchanged inbound messages results in a wasted update. Once the `SendMessages` operation

Algorithm 3.6: Parallel Wildfire Algorithm Using Atomics

```
$\sigma \leftarrow$ Arbitrary permutation on $\{1, \ldots, |V|\}$
Atomic integer $i \leftarrow 1$
do in parallel
    while Not Converged do
        // Reads the value of $i$ and increments $i$ atomically
        $j = AtomicFetchAndIncrement(i)$
        // Get the vertex to update
        $v = \sigma(j \mod |V|)$
        if New Messages into $v$ then
            Grab neighborhood locks in canonical order
            forall $j \in \mathcal{N}_v$ in the neighbors of $v$ do in parallel
                Compute Message $m^{(\text{new})}_{v \to j}$ using $\left\{m^{(\text{old})}_{i \to v}\right\}_{i \in \mathcal{N}_v}$
                if $\left|\left|m^{(new)}_{v \to j} - m^{(old)}_{v \to j}\right|\right|_1 > \beta$ then
                    Send message $m^{(\text{new})}_{v \to j}$
            Release locks
```

Algorithm 3.7: Residual Belief Propagation Algorithm

```
Priority Queue $Q$
Initialize $Q$ with all vertices at $\infty$ priority
do in parallel
    while TopResid($Q$) $> \beta$ do
        $v$ = Top (Q) // Get vertex with highest residual
        SendMessages($v$)
        Set $r_v = 0$ and update neighbors $r_{\mathcal{N}_v}$
```

is applied to a vertex, its residual is set to zero and its neighbors residuals are updated. Vertex scheduling has the advantage over message residuals of using all of the most recent information when updating a message.

The parallel Residual BP algorithm in Alg. 3.7 uses a priority queue to store the vertices in residual order. Each processor executes SendMessages on the next highest priority vertex from Q and then updates the priorities on all neighboring vertices.

When the message calculations are fairly simple, the priority queue becomes the central synchronizing bottleneck. An efficient implementation of a parallel priority queue is therefore the key to performance and scalability. There are numerous parallel priority queue algorithms in the literature [Crupi et al., 1996, Driscoll et al., 1988, Parberry, 1995, Sanders, 1998]. Many require sophisticated fine grained locking mechanisms while others employ binning strategies with constraints on the range and distribution of the priorities. Because the residual priorities are a heuristic, we find that relaxing the strict ordering requirement can further improve performance by reducing priority queue contention. In our implementation we randomly assigned vertices to priority queues associated with each processor. Each processor then draws from its own queue but can update the priorities of vertices owned by other processors.

3.6.1 Limitations of Message Residuals

The message centric residual defined in Eq. (3.14) may lead to nonuniform convergence in the posterior marginals (beliefs) and starvation of low-degree vertices. In this section we characterize these limitations and define a new residual measure Eq. (3.17) that directly captures changes in the estimated marginals.

Because the residual definition in Eq. (3.14) only considers the largest message change for each vertex, high-degree vertices with many small message changes may satisfy the termination condition while being far from convergence.

Lemma 3.6.1. *A small ϵ changes in all messages to a vertex of degree d can align resulting in a large $d\epsilon$ change in the belief.*

Proof. Consider a discrete binary random variable X_i with $d = |\mathcal{N}_i|$ and incoming messages $\{m_1, \ldots, m_d\}$ which are all uniform. Then the belief at that variable is also uniform (i.e., $b_i = [\frac{1}{2}, \frac{1}{2}]$). Suppose we then perturb each message by ϵ such that $m'_k(0) = \frac{1}{2} - \epsilon$ and $m'_k(1) = \frac{1}{2} + \epsilon$. The perturbed messages have changed by less than 2ϵ (i.e., $\forall k : ||m'_k - m_k||_1 \leq 2\epsilon$) and by the classic convergnec criterion Eq. (3.3) all messages have effectively converged. However the new belief is:

$$b'_i(0) = \frac{\left(\frac{1}{2} - \epsilon\right)^d}{\left(\frac{1}{2} + \epsilon\right)^d + \left(\frac{1}{2} - \epsilon\right)^d}.$$

The L_1 change of the belief due to the compounded ϵ change in each message is then:

$$\left\|b'_i(0) - b_i(0)\right\|_1 = \frac{1}{2} - \frac{\left(\frac{1}{2} - \epsilon\right)^d}{\left(\frac{1}{2} + \epsilon\right)^d + \left(\frac{1}{2} - \epsilon\right)^d}.$$

By constructing a 2^{nd} order Taylor expansion around $\epsilon = 0$ we obtain:

$$\left\|b'_i(0) - b_i(0)\right\|_1 \approx d\epsilon + O(\epsilon^3).$$

Therefore, the change in belief varies linearly in the degree of the vertex enabling small ϵ message residuals to translate into large $d\epsilon$ belief residuals. □

Alternatively, a large change in a single message may result in a small change in belief at a high degree vertex resulting in the over-scheduling.

Lemma 3.6.2. *A large $1 - \epsilon$ residual in a single message may result in a small ϵ change in belief at a high degree vertex.*

Proof. Consider the set $\{m_1, \ldots, m_d\}$ of binary messages with value $[1 - \epsilon, \epsilon]$ then the resulting belief at that variable would be

$$b_i(0) = \frac{(1 - \epsilon)^d}{(1 - \epsilon)^d + \epsilon^d}.$$

If we then change m_1 to $[\epsilon, 1-\epsilon]$ then the resulting belief is

$$b_i'(0) = \frac{(1-\epsilon)^{d-1}\epsilon}{(1-\epsilon)^{d-1}\epsilon + \epsilon^{d-1}(1-\epsilon)}.$$

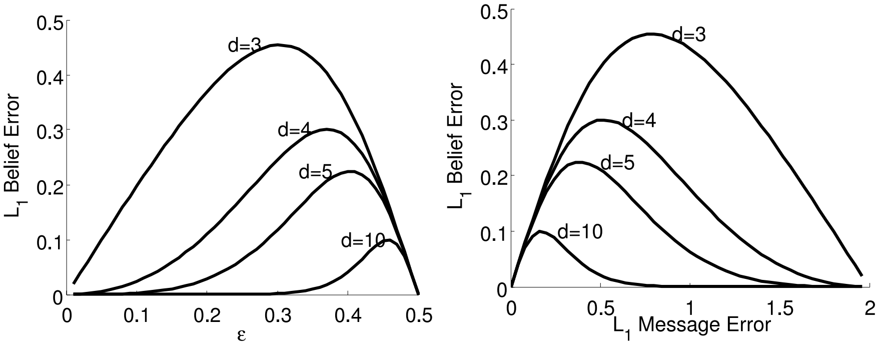
Assuming that $0 < \epsilon \leq \frac{1}{4}$ and $d \geq 3$,

$$\begin{aligned}
\frac{1}{2}\left\|b_i' - b_i\right\|_1 &= \frac{(1-\epsilon)^d}{(1-\epsilon)^d + \epsilon^d} - \frac{(1-\epsilon)^{d-1}\epsilon}{(1-\epsilon)^{d-1}\epsilon + \epsilon^{d-1}(1-\epsilon)} \\
&\leq 1 - \frac{(1-\epsilon)^{d-1}\epsilon}{(1-\epsilon)^{d-1}\epsilon + \epsilon^{d-1}(1-\epsilon)} \\
&= 1 - \frac{(1-\epsilon)^{d-2}}{(1-\epsilon)^{d-2} + \epsilon^{d-2}} \\
&= \frac{\epsilon^{d-2}}{(1-\epsilon)^{d-2} + \epsilon^{d-2}}
\end{aligned}$$

We can bound the denominator to obtain:

$$\frac{1}{2}\left\|b_i' - b_i\right\|_1 \leq \frac{\epsilon^{d-2}}{(1/2)^{d-3}} = (2\epsilon)^{d-3}\epsilon \leq \frac{\epsilon}{2^{d-3}}$$

In Figure 3.6a we plot the L_1 error in the belief $||b_i' - b_i||_1$, varying the value of ϵ. In Figure 3.6b, we plot the L_1 error in the beliefs against the L_1 change in m_1. The curves look nearly identical (but flipped) since the L_1 change is exactly $2 - 4\epsilon$ (recall that m_1 was changed from $[1-\epsilon, \epsilon]$ to $[\epsilon, 1-\epsilon]$).



(a) Belief Error against ϵ

(b) Belief Error against Message Error

Figure 3.6: (a): Given a vertex of degree d with all incoming messages equal to $[1-\epsilon, \epsilon]$. This graph plots the L1 change of belief on a vertex of degree d caused by changing one message to $[\epsilon, 1-\epsilon]$. (b): Similar to (a), but plots on the X-axis the L1 change of the message, which corresponds to exactly $2 - 4\epsilon$.

$\square$

3.6.2 Belief Residuals

The goal of belief propagation is to estimate the belief for each vertex. However, message scheduling and convergence assessment uses the change in messages rather than beliefs. Here we define a belief residual which addresses the problems associated with the message-centric approach and enables improved scheduling and termination assessment.

A natural definition of the belief residuals analogous to the message residuals defined in Eq. (3.13) is

$$r_j = \left|\left|b_i^{\text{new}} - b_i^{\text{old}}\right|\right|_1 \tag{3.15}$$

where b_i^{old} is the belief at vertex i the last time vertex i was updated. Unfortunately, Eq. (3.15) has a surprising flaw that admits premature convergence on acyclic graphical models even when the termination condition:

$$\max_{i \in V} \left|\left|b_i^{\text{new}} - b_i^{\text{old}}\right|\right|_1 \leq \beta \tag{3.16}$$

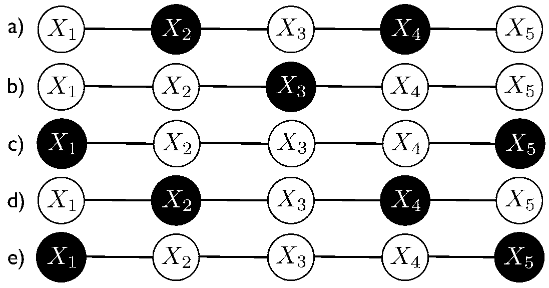
is set to $\beta = 0$.

Example 3.6.3 (Premature Termination Using Eq. (3.16))**.** *To see how Eq. (3.15) can lead to premature termination consider a chain MRF with five vertices and the binary factors $f_{X_i,X_{i+1}}(x_i, x_{i+1}) = I[x_i = x_{i+1}]$ and unary factors:*

$$f_{X_1} = \left[\tfrac{1}{9}, 9\right] \quad f_{X_2} = \left[\tfrac{9}{10}, \tfrac{1}{10}\right] \quad f_{X_3} = \left[\tfrac{1}{2}, \tfrac{1}{2}\right]$$

$$f_{X_4} = \left[\tfrac{1}{10}, \tfrac{9}{10}\right] \quad f_{X_5} = \left[9, \tfrac{1}{9}\right]$$

We begin by initializing all vertex residuals to infinity, and all messages to uniform distributions and then perform the following update sequence marked in black: After stage (b), $m_{2\to3} = f_{X_2}$ and $m_{4\to3} = f_{X_4}$. Since $b_{X_3} \propto (m_{2\to3} \times m_{4\to3}) \propto [1,1]$, the belief at X_3 X_3 will have uniform belief. Since X_3 is just updated, it will have a residual of 0. After stage (d), $m_{2\to3} \propto (f_{X_1} \times f_{X_2}) \propto f_{X_4}$ and $m_{4\to3} \propto (f_{X_5} \times f_{X_4}) \propto f_{X_2}$. These two messages therefore have swapped values since stage (b). Since $b_{X_3} \propto (m_{2\to3} \times m_{4\to3})$, the belief at X_3 will not be changed and will therefore continue to have uniform belief and zero residual. At this point X_2 and X_4 also have zero residual since they were just updated. After stage (e), the residuals at X_1 and X_5 at set to 0. However, the residuals on X_2 and X_4 remain zero since messages $m_{1\to2}$ and $m_{5\to4}$ will not change since state (c). All variables therefore now have a residual of 0. By Eq. (3.15) with $\beta = 0$ we have converged prematurely since no sequence of messages connects X_1 and X_5. The use of the naive belief residual in Eq. (3.15) will therefore converge to an erroneous solution.



An alternative formulation of the belief residual which does not suffer from premature termination and offers additional computational advantages is given by:

$$r_i^{(t)} \leftarrow r_i^{(t-1)} + \left|\left|b_i^{(t)} - b_i^{(t-1)}\right|\right|_1 \tag{3.17}$$

$$b_i^{(t)}(x_i) \propto \frac{b_i^{(t-1)}(x_i) m_{i\to j}^{(t)}(x_i)}{m_{i\to j}^{(t-1)}(x_i)}. \tag{3.18}$$

$b_i^{(t-1)}$ is the belief after incorporating the previous message and $b_i^{(t)}$ is the belief after incorporating the new message. Intuitively, the belief residual defined in Eq. (3.17) measures the cumulative effect of all message updates on the belief. As each new message arrives, the belief can be efficiently recomputed using Eq. (3.18). Since with each message change we can quickly update the local belief using Eq. (3.18) and corresponding belief residual (Eq. (3.17)), we do not need to retain the old messages and beliefs reducing the storage requirements and the overhead associated with scheduling.

The belief residual may also be used as the convergence condition:

$$\max_{i \in V} r_i \leq \beta \tag{3.19}$$

terminating when all vertices have a belief residual below a fixed threshold $\beta \geq 0$. Since Eq. (3.17) satisfies the triangle inequality, it is an upper bound on the total change in belief ensuring that the algorithm does not change while their are beliefs that have changed by more than β. Because Eq. (3.17) accumulates the change in belief with each new message, it will not lead to premature termination scenario encountered in the more naive belief residual definition (Eq. (3.15)).

Since the belief residual of a vertex can only increase until the vertex is updated and all vertices with belief residuals above the termination threshold are eventually updated, the belief residual prevents starvation. In addition, belief residuals also address the issue of over-scheduling in high degree vertices since a large change in a single inbound message that does not contribute to a significant change in belief will not significantly change the belief residual.

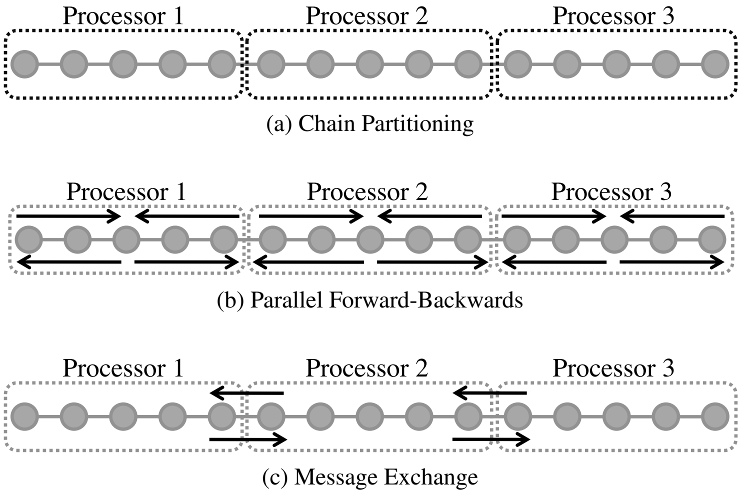
We use the belief residual for termination assessment and as a priority for Wildfire and Residual BP.

3.7 Splash Belief Propagation

The Round Robin BP, WildFire BP, and Residual BP algorithms improve upon the Synchronous BP algorithm. The Round Robin BP algorithm introduces *asynchronous* execution allowing each message update to access the most recent version of its dependent messages and eliminating the need for global barriers. The WildFire BP algorithm improves upon the Round Robin algorithm by eliminating wasted computation as messages converge at different rates. Finally the Residual BP algorithm focuses computation resources where they can make the most progress further accelerating convergence. Unfortunately, despite all of these improvements, all the algorithms we have described so far share the same asymptotic $O(n\tau_\epsilon/p + \tau_\epsilon)$ performance on chain graphical models. The problem is none of these algorithms directly address the forward-backward sequential structure of inference.

3.7.1 Optimal Parallel Scheduling on Chains

We begin by developing a simple algorithm (Alg. 3.8) that composes the forward-backward schedule to efficiently construct a τ_ϵ-approximation in parallel and achieve the asymptotic lower bound given in Theorem 3.3.2. Alg. 3.8 begins by dividing the chain evenly among the p processors as shown in Figure 3.7a. Then, in parallel, each processor runs the sequential forward-backward algorithm on its sub-chain as shown in Figure 3.7b. Finally each processor exchanges messages along the boundaries and repeats the procedure until convergence. Notice, the algorithm does not require knowledge of τ_ϵ and instead relies on the convergence criterion to ensure an accurate approximation.



(a) Chain Partitioning

(b) Parallel Forward-Backwards

(c) Message Exchange

Figure 3.7: An illustration of the optimal scheduling on a chain. **(a)** The chain is partitioned evenly among the processors. **(b)** In parallel each processor runs the forward-backward sequential scheduling. The forward sweep updates messages towards the center vertex and the backwards sweep updates messages towards the boundary. This slightly abnormal forward-backward sweep will be more natural when we generalize it to arbitrary cyclic graphs in the Parallel Splash Algorithm. **(c)** Messages that cross the processor boundaries are exchanged and steps **(b)** and **(c)** are repeated until convergence.

We now show that this simple procedure achieves the asymptotic lower bound:

Theorem 3.7.1 (Optimality of Alg. 3.8)**.** *Given a chain graph with n vertices and $p \leq n$ processors, the ChainSplash belief propagation algorithm, achieves a τ_ϵ level approximation for all vertices in time*

$$O\left(\frac{n}{p} + \tau_\epsilon\right)$$

Proof. As with the lower bound proof we will consider messages in only one direction from x_i to x_{i+1}. After each local execution of the forward-backward algorithm all vertices become left-aware of all vertices within each processor block. After messages are exchanged the left most vertex becomes left-aware of the right most vertex on the preceding processor. By transitivity of left-awareness, after k iterations, all vertices become left-aware of $(k-1)n/p$ vertices. We require all vertices to be made τ_ϵ left-aware and so solving for k-iterations we obtain:

$$\begin{aligned} \tau_\epsilon &= (k-1)\frac{n}{p} \\ k &= \frac{\tau_\epsilon p}{n} + 1 \end{aligned}$$

Each iteration is executed in parallel in time n/p so the total running time is then:

$$\begin{aligned} \frac{n}{p}k &= \frac{n}{p}\left(\frac{\tau_\epsilon p}{n} + 1\right) \\ &= \frac{n}{p} + \tau_\epsilon \end{aligned}$$

□

Algorithm 3.8: ChainSplash BP Algorithm

```
Input: Chain graphical model $(V, E)$ with $n$ vertices
// Partition the chain over the $p$ processors
forall $i \in \{1, \ldots, p\}$ do in parallel
    $B_i \leftarrow \left\{x_{\lceil (i-1)n/p \rceil}, \ldots, x_{\lceil in/p \rceil - 1}\right\}$
while Not Converged do
    // Run Forward-Backward Belief Propagation on each Block
    forall $i \in \{1, \ldots, p\}$ do in parallel
        Run Sequential Forward-Backward on $B_i$
    // Exchange Messages
    forall $i \in \{1, \ldots, p\}$ do in parallel
        Exchange messages with $B_{i-1}$ and $B_{i+1}$
```

Our new Splash BP algorithm generalizes Alg. 3.8 to arbitrary cyclic graphical models. The Splash algorithm is composed of two core components, the Splash operation which generalizes the forward-backward schedule to arbitrary cyclic graphical models, and a dynamic Splash scheduling heuristic which ultimately determines the shape, size, and location of the Splash operation. We will first present the Parallel Splash algorithm as a sequential algorithm, introducing the Splash operation, belief scheduling, and how they are combined to produce a simple single processor Splash algorithm. Then in subsequent sections we describe the parallel structure of this algorithm and how it can efficiently be mapped to shared and distributed memory systems.

3.7.2 The Splash Operation

The Splash BP algorithm is built around the Splash[3] operation (Alg. 3.9 and Figure 3.8) which generalizes the forward-backward schedule illustrated in Figure 3.2a by constructing a small sub-tree and then executing a local forward-backward schedule on the sub-tree. By scheduling message calculations along a local tree, we directly address the sequential message dependencies and retain the optimal forward-backward schedule when applied to acyclic models.

The inputs to the Splash operation are the root vertex v and the maximum allowed size of the Splash, W. The Splash operation begins by constructing a sub-tree rooted at v, adding vertices in breadth first order until the sub-tree exceeds W which is measured in terms of number of floating point operations associated with executing BP inference on the sub-tree. The size of each sub-tree is restricted to ensure work balance across processors, which will execute multiple Splash operations in parallel. The work associated with each vertex u (which could be a variable or factor) depends on the size of the messages and the number of neighbors in the graph:

$$w_u = |\mathcal{N}_u| \times |\mathcal{X}_u| + \sum_{X_v \in \mathcal{N}_u} |\mathcal{X}_v| , \tag{3.20}$$

where $|\mathcal{N}_u| \times |\mathcal{X}_u|$ represents the work required to recompute all outbound messages and $\sum_{X_v \in \mathcal{N}_u} |\mathcal{X}_v|$ is the work required to update the beliefs of all the neighboring vertices (needed for dynamic scheduling). In

[3] The algorithm is called Splash BP because the forward-backward "wavefront" seen when running the algorithm on large planar graphs looks like a splash.

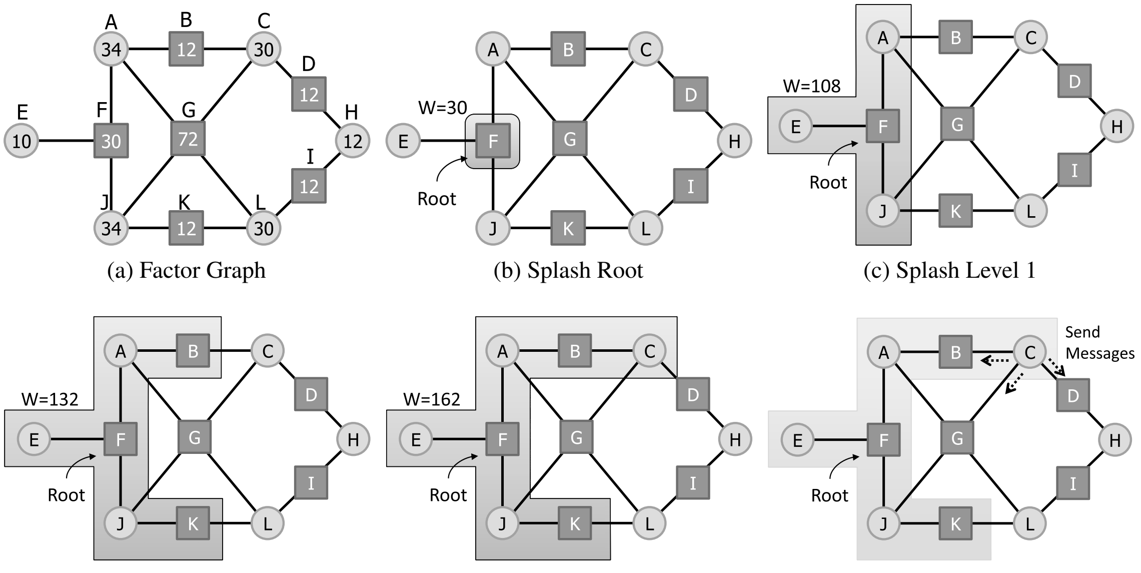
Algorithm 3.9: Splash(v, W)

```
  Input: vertex $v$
  Input: maximum Splash size $W$
  // Construct the breadth first search ordering with $W$ message
     computations and rooted at $v$.
  fifo $\leftarrow$ [] // FiFo Spanning Tree Queue
  $\sigma \leftarrow (v)$ // Initial Splash ordering is the root $v$
  AccumW $\leftarrow w_v$ // Accumulate the root vertex work
  visited $\leftarrow \{v\}$ // Set of visited vertices
  fifo.Enqueue($\mathcal{N}_v$)
  while fifo is not empty do
      $u \leftarrow$ fifo.Dequeue()
      // If adding $u$ does not cause Splash to exceed limit
      if AccumW + $w_u \leq W$ then
          AccumW $\leftarrow$ AccumW + $w_u$
          Add $u$ to the end of the ordering $\sigma$
          foreach neighbors $v \in \mathcal{N}_u$ do
              if $v$ is not in visited then
                  fifo.Enqueue($v$) // Add to boundary of spanning tree
                  visited $\leftarrow$ visited $\cup \{v\}$ // Mark Visited
  // Forward Pass: sends messages from the leaves to root
1 foreach $u \in$ ReverseOrder($\sigma$) do
      SendMessages($u$) // Update priorities if necessary.
  // Backward Pass: sends messages from the root to leaves
2 foreach $u \in \sigma$ do
      SendMessages($u$) // Update priorities if necessary.
```

Table 3.1 we compute the work associated with each vertex shown in Figure 3.8a.

The local spanning tree rooted at vertex F in Figure 3.8 is depicted by the shaded rectangle which grows outwards in the sequence of figures Figure 3.8b through Figure 3.8e. The maximum splash size is set to $W = 170$. In Figure 3.8b the Splash contains only vertex F (the root) and has total accumulated work of $w = 30$. The vertices A, E, and F are added in Figure 3.8c expanding the breadth first search without exceeding $W = 170$. The effect of $W = 170$ is seen in Figure 3.8d vertices B and K are added but vertex G is *skipped* because including G would cause the Splash to exceed $W = 170$. The maximum splash size is achieve in Figure 3.8e when vertex C is added and no other vertices may be added without exceeding $W = 170$. The final splash ordering is $\sigma = [F, E, A, J, B, K, C]$.

Using the *reverse* of the breadth first search ordering constructed in the first phase, the SendMessages operation is sequentially invoked on each vertex in the spanning tree (Line 1) starting at the leaves, in analogy to the forward sweep. The function SendMessages(v) updates all outbound messages from vertex v using the most recent inbound messages. In Figure 3.8f, SendMessages(C) is invoked on vertex C causing the messages $m_{C\to B}$, $m_{C\to G}$, and $m_{C\to D}$ to be recomputed. Finally, messages are computed in the original σ ordering starting at the root and invoking SendMessages sequentially on



(a) Factor Graph (b) Splash Root (c) Splash Level 1

(d) Splash Level 2 (e) Splash Level 3 (f) SendMessages

Figure 3.8: A splash of splash size $W = 170$ is grown starting at vertex F. The Splash spanning tree is represented by the shaded region. **(a)** The initial factor graph is labeled with the vertex work (w_i) associated with each vertex. **(b)** The Splash begins rooted at vertex F with accumulated work $w = 30$. **(c)** The neighbors of F are added to the Splash and the accumulated work increases to $w = 108$. **(d)** The Splash expands further to include vertex B and K but does not include vertex G because doing so would exceed the maximum splash size of $W = 170$. **(e)** The splash expand once more to include vertex C but can no longer expand without exceeding the maximum splash size. The final Splash ordering is $\sigma = [F, E, A, J, B, K, C]$. **(f)** The SendMessages operation is invoked on vertex C causing the messages $m_{C \to B}$, $m_{C \to G}$, and $m_{C \to D}$ to be recomputed.

each vertex, completing the backwards sweep. Hence, with the exception of the root vertex v, all messages originating from each vertex in σ are computed twice, once on the forwards sweep in Line 1 and once in the backwards sweep in Line 2.

It is easy to see that by repeatedly executing p parallel Splashes (see Figure 3.7) of size $W = wn/p$ (where w is the work of updating a single vertex) placed evenly along the chain we can recover the *optimal* algorithm Alg. 3.8.

3.7.3 The Sequential SplashBP Algorithm

By combining the Splash operation with the belief residual scheduling to select Splash roots we obtain the sequential version of the SplashBP algorithm given in Alg. 3.10. The belief residual of each vertex is initialized to the largest possible value (e.g., `Inf`) to ensure that every vertex is updated at least once. The Splash operation is repeatedly applied to the vertex with highest priority. During the Splash operation the priorities (belief residuals) of all vertices in the Splash and along the boundary are updated: after `SendMessages` is invoked on a vertex its belief residual is set to zero and the priority of its neighboring vertices is increased by the change in their beliefs.

Vertex	Assignments (A_i)	Degree ($\lvert\mathcal{N}_i\rvert$)	Neighbor Costs $\left(\sum_{w\in\mathcal{N}_i} \lvert\mathcal{N}_w\rvert\right)$	Work (w_i)
A	2	3	$2^3 + 2^4 + 2^2$	34
B	2^2	2	$2 + 2$	12
C	2	3	$2^2 + 2^4 + 2^2$	30
D	2^2	2	$2 + 2$	12
E	2	1	2^3	10
F	2^3	3	$2 + 2 + 2$	30
G	2^4	4	$2 + 2 + 2 + 2$	72
H	2	2	$2^2 + 2^2$	12

Table 3.1: The work associated with each vertex in Figure 3.8a is computed using Eq. (3.20). We have omitted vertices I, J, K, and L sinc they are equivalent to vertices D, A, B, and C respectively.

Algorithm 3.10: The Sequential Splash Algorithm

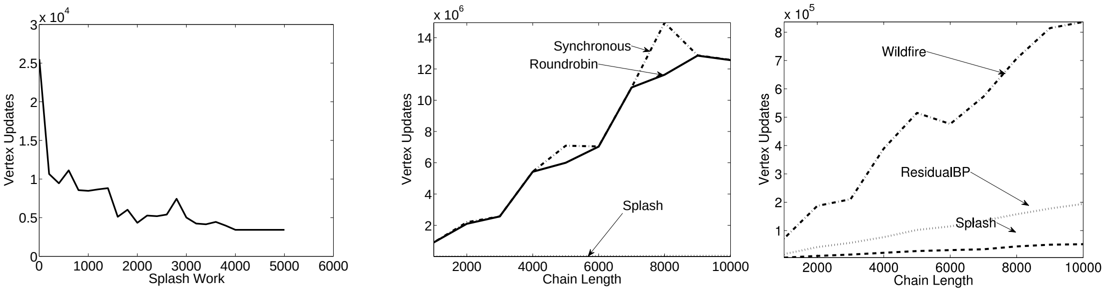
```
Input: Constants: maximum Splash size $W$, termination bound $\beta$
$Q \leftarrow$ InitializeQueue($Q$)
Set All Residuals to $\infty$
while TopResid($Q$) $> \beta$ do
    $v \leftarrow$ Top($Q$)
    Splash($Q, v, W$) // Priorities updated after SendMessages
```

3.7.4 Evaluating SplashBP in the Sequential Setting

To demonstrate the performance gains of the sequential SplashBP algorithm on chain structured models we constructed a set of synthetic chain graphical models and evaluated the running time on these models for a fixed convergence criterion while scaling the size of the Splash in Figure 3.9a and while scaling the size of the chain in Figure 3.9b. Each chain consists of binary random variables with weak random node potentials and strong attractive edge potentials. As the size of the Splash increases (Figure 3.9a) the total number of updates on the chain decreases reflecting the optimality of the underlying forward-backward structure of the Splash.

In Figure 3.9b we compare the running time of Splash with Synchronous, Round-Robin, Residual, and Wildfire BP algorithms as we increase the size of the model. The conventional Synchronous and Round-Robin algorithms are an order of magnitude slower than Wildfire, ResiduaBP, and Splash and scale poorly forcing separate comparisons for readability. Nonetheless, in all cases the Splash algorithm is substantially faster and demonstrates better scaling with increasing model size.

The running time, computational efficiency, and accuracy of the SplashBP algorithm were evaluated in the single processor setting. In Figure 3.10a, Figure 3.10b, and Figure 3.10c we plot the average belief accuracy, worst case belief accuracy, and prediction accuracy against the runtime on a subset of the UAI 2008 Probabilistic Inference Evaluation dataset [Darwiche et al., 2008]. We ran each algorithm to $\beta = 10^{-5}$ convergence and report the runtime in seconds and the marginal estimates for all variables. We compared against the exact marginals obtained using Ace 2.0 [Huang et al., 2006]. In all cases the Splash algorithm obtained the most accurate belief estimates in the shortest time. The dynamical scheduled algorithms, Wildfire and Residual BP, consistently outperformed the statically scheduled algorithms, Round-Robin and Synchronous BP. We also assessed accuracy on a protein side chain prediction task [Yanover et al., 2007]



(a) Runtime versus Splash Size

(b) Runtime versus Chain Size

Figure 3.9: In this figure, we plot the number of vertex updates on a randomly generated chain graphical model. Run-time to convergence is measured in vertex updates rather than wall clock time to ensure a fair *algorithmic* comparison and eliminate hardware and implementation effects which appear at the extremely short run-times encountered on these simple models. **(a)** The number of vertex updates made by Sequential Splash BP, fixing the chain length to 1000 variables, and varying the Splash size. **(b)** The number of vertex updates made by various BP algorithms varying the length of the chain. Two plots are used to improve readability since the Splash algorithm is an order of magnitude faster than the Synchronous and Round-Robin algorithms however the Splash algorithm curve is the same for both plots.

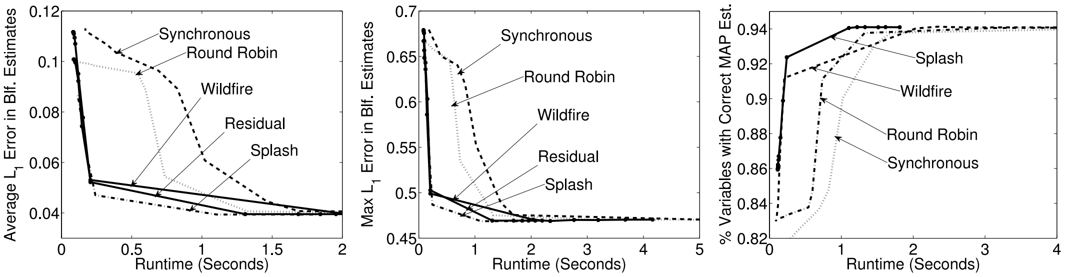
where the goal is to estimate the orientations of each side chain in a protein structure. Here we find that all belief propagation algorithms achieve roughly the same prediction accuracy of 73% for χ_1 and χ_2 angles which is consistent with the results of Yanover et al. [2007].

Not only does SplashBP converge faster than other belief propagation algorithms, it also converges more often. We assessed the convergence of the SplashBP algorithm using several different metrics. In Figure 3.11a we plot the number of protein networks that have converged ($\beta = 10^{-5}$) against the run-time. In Figure 3.11b we plot the number of protein networks that have converged against the number of message computations. Again, we see that SplashBP converges sooner than other belief propagation algorithms.

In Figure 3.12a and Figure 3.12b we directly compare the running time and work of the SplashBP algorithm against other common scheduling strategies on a set of challenging protein-protein interaction networks obtained from Taskar et al. [2004b]. The results are presented for each of the 8 networks separately with bars in the order *Splash*, *Residual*, *Wildfire*, *Round-Robin*, and *Synchronous*. All algorithms converged on all networks except residual belief propagation which failed to converge on network five. In all cases the Splash algorithm achieves the shortest running time and is the in the bottom two in total message computations. Since the Splash algorithm favors faster message computations (lower degree vertices), it is possible for the Splash algorithm is able to achieve a shorter runtime even while doing slightly more message computations.

3.7.5 Parallelizing the Splash Algorithm

We easily parallelize the SplashBP algorithm (Alg. 3.11) by executing multiple Splash operations in parallel generalizing Alg. 3.8. Each of the p processor executes it own separate Splash operation on the top p highest priority vertices.



(a) Average L_1 Error (b) Maximum L_1 Error (c) MAP Accuracy

Figure 3.10: We assessed the accuracy of Splash algorithm using the exact inference challenge networks from Darwiche et al. [2008] as well as the protein side chain prediction networks obtained from Yanover et al. [2007]. In **(a)** and **(b)** we plot the average and max L_1 error in the belief estimates for all variables as a function of the running time. In **(c)** we plot the prediction accuracy of the MAP estimates as a function of the running time. In all cases we find that Splash belief propagation achieves the greatest accuracy in the least time.

Algorithm 3.11: Parallel Splash Belief Propagation Algorithm

```
Input: Constants: maximum Splash size $W$, termination bound $\beta$
$Q \leftarrow$ InitializeQueue($Q$)
Set All Residuals to $\infty$
forall processors do in parallel
    while TopResidual(Q) $> \beta$ do
        $v \leftarrow$ Pop($Q$) // Atomic
        Splash($Q, v, W$) // Updates vertex residuals
        $Q$.Push(($v$, Residual($v$))) // Atomic
```

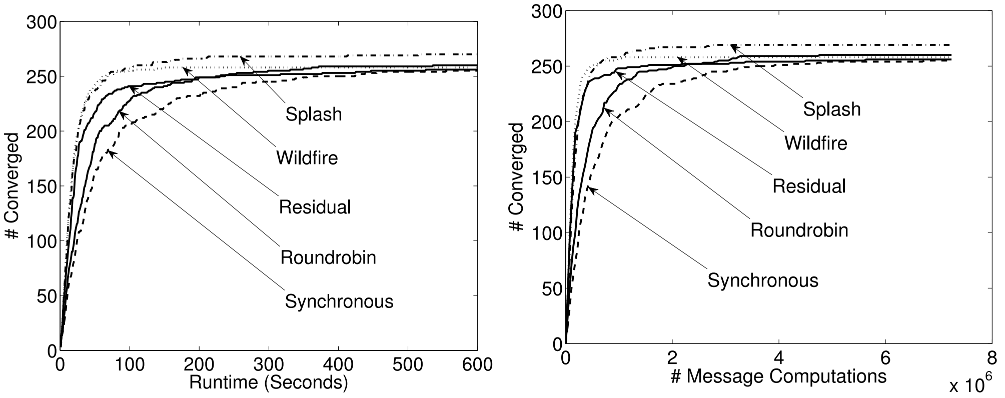
While we do not require that parallel Splash operations contain disjoint sets of vertices, we do require that each Splash has a unique root which is achieved through a shared priority queue and atomic `Push` and `Pop` operations.

To achieve maximum parallel performance the Parallel Splash algorithm relies on an efficient parallel scheduling queue to minimize processor locking and sequentialization when `Push`, `Pop`, and `UpdatePriority` are invoked. While there is considerable work from the parallel computing community on efficient parallel queue data structures, we find that in the shared memory setting basic locking queues provide sufficient performance. In the distributed Splash algorithm discussed in Section 3.8 we employ a distributed scheduling queue in a work balanced manner to eliminate processor contention.

3.7.6 Parallel Splash Optimality on Chains

We now show that the parallel SplashBP algorithm using belief residual scheduling achieves the optimal running time for chain graphical models. We begin by relating the Splash operation to the vertex residuals.

Lemma 3.7.2 (Splash Residuals)**.** *Immediately after the Splash operation is applied to an acyclic graph*



(a) Protein Convergence vs. Time

(b) Protein Convergence vs. Work

Figure 3.11: The Splash algorithm demonstrates faster and more consistent convergence than other baseline algorithms on a single processor. In the number of converged ($\beta = 10^{-5}$) networks (out of 276) is plotted against the runtime **(a)** and number of message calculations **(b)**.

all vertices interior to the Splash have zero belief residual.

Proof. The proof follows naturally from the convergence of BP on acyclic graphs. The Splash operation runs BP to convergence on the *sub-tree* contained within the Splash. Therefore the marginal estimates at each variable will have converged and the corresponding belief residuals will be zero. □

After a Splash is completed, the residuals associated with vertices interior to the Splash are propagated to the exterior vertices along the boundary of the Splash. Repeated application of the Splash operation will continue to move the boundary residual leading to Lemma 3.7.3.

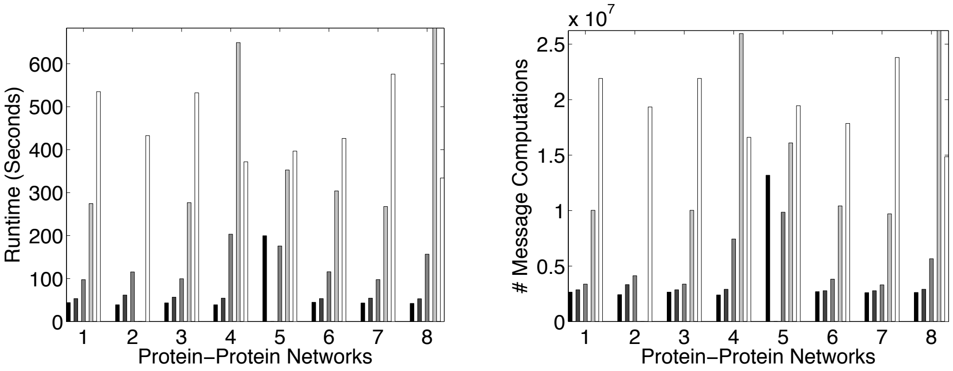
Lemma 3.7.3 (Basic Convergence)**.** *Given a chain graph where only one vertex has nonzero residual, the Splash algorithm with Splash size W will run in $O(\tau_\epsilon + W)$ time.*

Proof. When the first Splash, originating from the vertex with nonzero residual is finished, the interior of the Splash will have zero residual as stated in Lemma 3.7.2, and only the boundary of the Splash will have non-zero residual. Because all other vertices initially had zero residual and messages in opposite directions do not interact, each subsequent Splash will originate from the boundary of the region already covered by the previous Splash operations. By definition the convergence criterion is achieved after the high residual messages at the originating vertex propagate a distance τ_ϵ. However, because the Splash size is fixed, the Splash operation may propagate messages an additional W vertices. □

If we set the initial residuals to ensure that the first p parallel Splashes are uniformly spaced, SplashBP obtains the optimal lower bound.

Theorem 3.7.4 (SplashPB Optimality on Chain Models)**.** *Given a chain graph with n vertices and $p \leq n$ processors we can apply the Splash algorithm with the Splash size set to $W = n/p$ and uniformly spaced initial Splashes to obtain a τ_ϵ-approximation in expected running time*

$$O\left(\frac{n}{p} + \tau_\epsilon\right)$$



(a) Runtime

(b) Work

Figure 3.12: We assessed runtime and convergence on eight different protein-protein interaction networks obtained from Taskar et al. [2004b]. In **(a)** and **(b)** we plot the runtime in seconds and the work in message computations for each algorithm on each network. Each bar represents a different algorithm in the order *Splash*, *Residual*, *Wildfire*, *Round-Robin*, and *Synchronous* from left (darkest) to right (lightest).

Proof of Theorem 3.7.4. We set every n/p vertex $\left\{X_{n/2p}, X_{3n/2p}, X_{5n/2p}, \ldots\right\}$ to have slightly higher residual than all other vertices forcing the first p Splash operations to start on these vertices. Since the size of each splash is also $W = n/p$, all vertices will be visited in the first p splashes. Specifically, we note that at each Splash only produces two vertices of non-zero residual (see Lemma 3.7.2). Therefore there are at most $O(p)$ vertices of non-zero residual left after the first p Splashes.

To obtain an upper bound, we consider the runtime obtained if we independently compute each τ_ϵ subtree rooted at a vertex of non-zero residual. This is an upper bound as we observe that if a single Splash overlaps more than one vertex of non-zero residual, progress is made simultaneously on more than one subtree and the running time can only be decreased.

From Lemma 3.7.2, we see that the total number of updates needed including the initial $O(p)$ Splash operations is $O(p(\tau_\epsilon + W)) + O(n) = O(n + p\tau_\epsilon)$. Since work is evenly distributed, each processor performs $O(n/p + \tau_\epsilon)$ updates. □

In practice, when the graph structure is not a simple chain graph, it may be difficult to evenly space Splash operations. By randomly placing the initial Splash operations we can obtain a factor $\log(p)$ approximation in expectation.

Corollary 3.7.5 (SplashBP with Random Initialization)**.** *If all residuals are initialized to a random value greater than the maximum residual, the total expected running time is at most* $O(\log(p)(n/p + \tau_\epsilon))$.

Proof. Partition the chain graph into p blocks of size n/p. If a Splash originates in a block then it will update all vertices interior to the block. The expected time to Splash (collect) all p blocks is upper bounded[4] by the coupon collectors problem. Therefore, at most $O(p\log(p))$ Splash operations (rather than the p Splash operations used in Theorem 3.7.4) are required in expectation to update each vertex at least once. Using the same method as in Theorem 3.7.4, we observe that the running time is $O(\log(p)(n/p + \tau_\epsilon))$. □

[4] Since candidate vertices are removed between parallel rounds, there should be much fewer collection steps than the analogous coupon collectors problem.

Algorithm 3.12: DynamicSplash(Q, v, W)

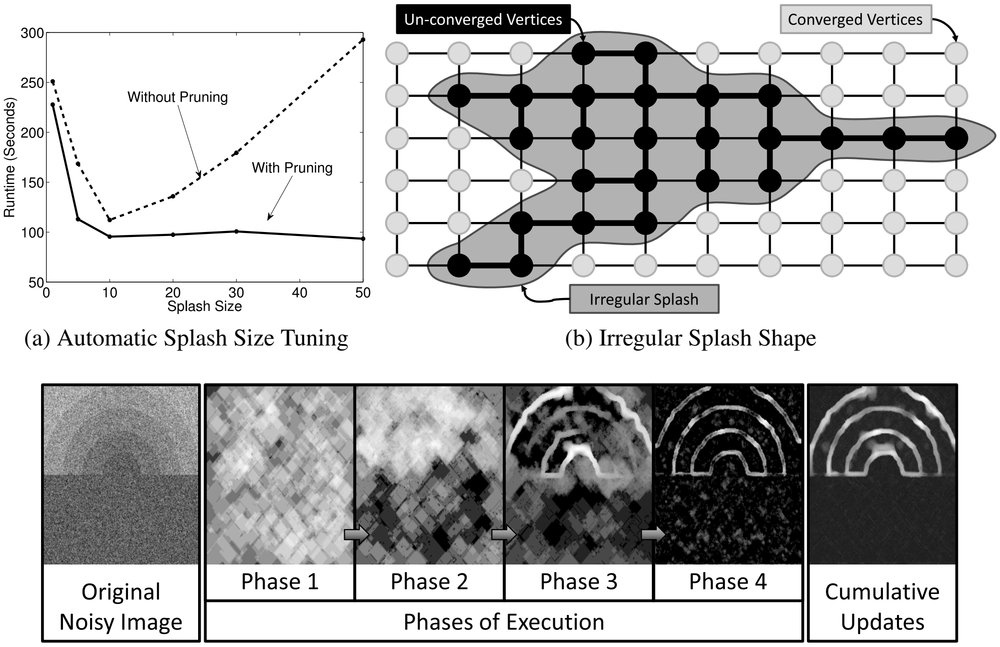
```
Input: scheduling queue $Q$, vertex $v$, maximum splash size $W$
fifo ← [] // FiFo Spanning Tree Queue
$\sigma \leftarrow (v)$ // Initial Splash ordering is the root $v$
AccumW ← $w_v$ // Accumulate the root vertex work
visited ← $\{v\}$ // Set of visited vertices
fifo.Enqueue($\mathcal{N}_v$)
while fifo is not empty do
    $u \leftarrow$ fifo.Dequeue()
    if AccumW + $w_u \leq W$ then
        AccumW ← AccumW + $w_u$
        Add $u$ to the end of the ordering $\sigma$
        foreach neighbors $v \in \mathcal{N}_u$ do
            if Belief Residual of $v$ is greater than $\beta$ and is not in visited then
                fifo.Enqueue($v$) // Add boundary
                visited ← visited $\cup \{v\}$ // Mark Visited
foreach $u \in$ ReverseOrder($\sigma$) do
    SendMessages($u$)
    Q.SetPriority($u$, $0$) // Set Belief residual to zero
    foreach $v \in \mathcal{N}_u$ do Q.UpdatePriority($v$)
foreach $u \in \sigma$ do
    SendMessages($u$)
    Q.SetPriority($u$, $0$) // Set Belief residual to zero
    foreach $v \in \mathcal{N}_u$ do Q.UpdatePriority($v$)
```

3.7.7 Dynamically Tuning the Splash Operation

A weakness of the Splash operation is that it requires selecting the Splash size parameter W which affects the overall performance. If the Splash size is too large then the algorithm will be forced to recompute messages that have already converged. Alternatively, if the Splash size is set too small then we do not exploit the local sequential structure. To address this weakness introduced **Dynamic Splashes** which substantially improve performance in practice and eliminate the need to tune the Splash size parameter.

Dynamic Splashes (Alg. 3.12) automatically adjust the size and shape of each Splash by adopting the Wildfire BP heuristic in the sub-tree construction procedure of the Splash operation. During the initial breadth first search phase of the Dynamic Splash operation to excludes vertices with belief residuals below the termination condition. This ensures that we do not recompute messages that have already converged, and more importantly allows the Splash operation to *dynamically adapt* to the local convergence patterns in the graph. In addition, as the algorithm converges, low belief residuals automatically *prune* the breadth first search tuning the size of each Splash and eliminating the need to carefully choose the Splash size W. Instead we can fix Splash size to a relatively large fraction of the graph (e.g., n/p) and let pruning automatically decrease the Splash size as needed.

In Figure 3.13a we plot the running time of the Parallel Splash algorithm with different Splash sizes W



(a) Automatic Splash Size Tuning

(b) Irregular Splash Shape

(c) Execution of the Splash Algorithm on the Denoising Task

Figure 3.13: **(a)** The running time of the Splash algorithm using various different Splash sizes with and without pruning. **(b)** The vertices with high belief residual,shown in black, are included in the Splash while vertices with belief residual below the termination threshold, shown in gray are excluded. **(c)** To illustrate the execution of the Splash BP algorithm we ran it on a simple image denoising task and took snapshots of the program state at four representative points (phases) during the execution. The cumulative vertex updates (number of times SendMessages was invoked since the last snapshot) are plotted with brighter regions being updates more often than darker regions. Initially, large regular (rectangular) Splashes are evenly spread over the entire model. As the algorithm proceeds the Splashes become smaller and more irregular focusing on the *challenging* regions along the boundaries of the underlying image.

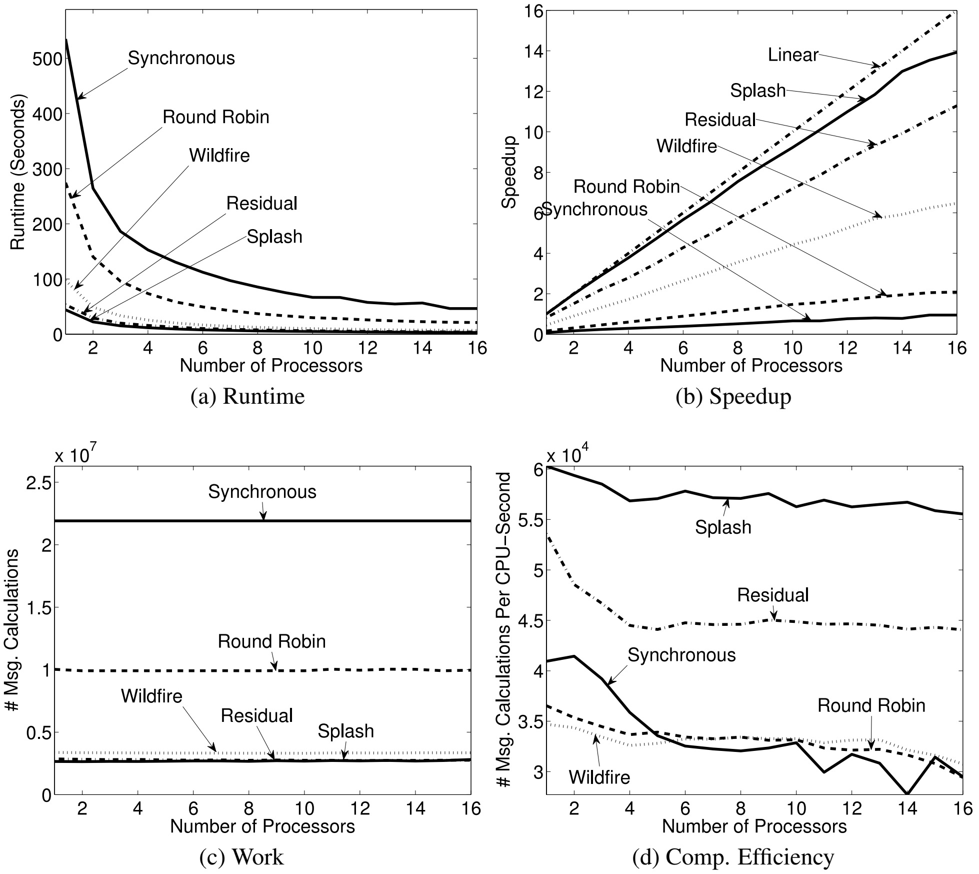
both with and without Splash pruning enabled. With Splash pruning disabled there is clear optimal Splash size. However with Splash pruning enabled, increasing the size of the Splash beyond the optimal size does not reduce the performance. In Figure 3.13c we illustrate examples of the Splashes at various phases of the algorithm on the image denoising task (see Figure 2.1). Initially the Splashes are relatively large and uniform but as the algorithm converges the Splashes shrink and adapt to the local shape of the remaining non-converged regions in the model.

3.7.8 Implementing SplashBP in the Shared Memory Setting

The parallel SplashBP algorithm presented in the previous section (Alg. 3.11) can easily be mapped to the shared memory setting described in Section 2.2.1. Because the shared memory setting ensures relatively low latency access to all memory it is not necessary[5] to partition the graph over the processors. However, because multiple processors can read and modify the same message, we must ensure that messages and

[5] It is possible that in an extreme NUMA setting partitioning could improve performance and in this case we advocate using ideas from the distributed implementation.

beliefs are manipulated in a consistent manner which is accomplished using the `SendMessages` locking scheme described in Alg. 3.5.



(a) Runtime

(b) Speedup

(c) Work

(d) Comp. Efficiency

Figure 3.14: Multicore results for a Protein-Protein Interaction Networks [Taskar et al., 2004a]

In Figure 3.14 we compare each algorithm in terms of runtime, speedup, work, and efficiency as a function of the number of cores. All experiments were run on protein-protein interaction networks obtained from Taskar et al. [2004b] with over 14K binary variables, 20K factors, and 50K edges. The runtime, shown in Figure 3.14a, is measured in seconds of elapsed wall clock time before convergence. On all of the networks the SplashBP algorithm is the fastest. The static scheduling algorithms (Round-Robin and Synchronous BP) are consistently slower than the dynamic scheduling algorithms (Residual and Widlfire BP).

The Splash algorithm demonstrates nearly linear scaling. In Figure 3.14b we plot the strong scaling of each algorithm measured relative to the fastest single processor algorithm. As a consequence of the relative-to-best scaling, inefficient algorithms may exhibit a speedup less than one. Furthermore, we again see a consistent pattern in which the dynamic scheduling algorithms dramatically outperform the static scheduling algorithms. The inefficiency in the static scheduling algorithms is so dramatic that the parallel variants seldom achieve more than a factor of 2 speedup using 16 processors.

We measured algorithm work in Figure 3.14c, in terms of the number of message calculations before convergence. The total work, which is a measure of algorithm efficiency, should be as small as possible and not depend on the number of processors. We find that the Splash algorithm generally does the least

Algorithm 3.13: The Distributed BP Algorithm

```
1  $\mathcal{B} \leftarrow$ Partition($G, p$) // Partition the graph over processors
   DistributeGraph($G, \mathcal{B}$)
   forall $b \in \mathcal{B}$ do in parallel
       repeat
           // Perform BP Updates according to local schedule
2          RunBPAlgorithmOn($b$)
           RecvExternalMsgs() // Receive and integrate messages
           SendExternalMsgs() // Transmit boundary messages
       until not Converged() // Distributed convergence test
```

work and that the number of processors has minimal impact on the total work done.

Finally, we assessed computation efficiency (Figure 3.14d) by computing the number of message calculations per processor-second. The computational efficiency is a measure of the raw throughput of message calculations during inference. Ideally, the efficiency should remain as high as possible. The computational efficiency, captures the cost of building spanning trees in the Splash algorithm or frequently updating residuals in the residual algorithm. The computational efficiency also captures concurrency costs incurred at locks and barriers. In all cases we see that the Splash algorithm is considerably more computationally efficient. While it is tempting to conclude that the implementation of the Splash algorithm is more optimized, all algorithms used the same message computation, scheduling, and support code and only differ in the core algorithmic structure. Hence it is surprising that the Splash algorithm, even with the extra overhead associated with generating the spanning tree in each Splash operations. However, by reducing the frequency of queue operations and the resulting lock contention, by increasing cache efficiency through message reuse in the forward and backward pass, and by favoring lower work high residual vertices in each spanning tree, the Splash algorithm is able to update more messages per processor-second.

3.8 Belief Propagation Algorithms on Distributed Architectures

In contrast to the shared memory setting where each processor has symmetric fast access to the entire program state, in the distributed memory setting each process can only directly access its local memory and must pass messages to communicate with other processors. Therefore efficient distributed parallel algorithms are forced to explicitly distribute the program state.

While the distributed memory setting can be more challenging, it offers the potential for linear scaling in memory capacity and memory bandwidth. To fully realize the linear scaling in memory capacity and bandwidth we must partition the program state in a way that places only a fraction $1/p$ of the global program state on each processor. Evenly distributing computation and storage while minimizes across machine coordination is the central challenge in the design of scalable distributed parallel algorithms. While the distributed setting often considers systems with network and processor failure, in this section we assume that the all resources remain available throughout execution and that all messages eventually reach their destination.

In Alg. 3.13 we present a generic distributed BP algorithm composed of partitioning phase (Line 1) after which each processor repeatedly executes a BP update (Line 2) using a local schedule followed by inter-processor communication and a distributed convergence test (Line 3). To achieve balance computation and communication we employ weighted graph partitioning and over-partitioning which we will now describe. As it is often impractical to maintain an exact schedules across distributed memory, relaxed versions are discussed in Section 3.8.2. Finally, we address the challenges of distributed convergence assessment in Section 3.8.3.

3.8.1 Partitioning the Factor Graph and Messages

To distribute the state of the program among p processors, we begin by partitioning the graphical model and BP messages (parameters). To maximize throughput and hide network latency, we must minimize inter-processor communication and ensure that the data needed for message computations are locally available. We define a partitioning of the factor graph over p processors as a set $\mathcal{B} = \{B_1, ..., B_p\}$ of disjoint sets of vertices $B_k \subseteq V$ such that $\cup_{k=1}^{p} B_k = V$. Given a partitioning $\mathcal{B}$ we assign all the factor data associated with $f_i \in B_k$ to the k^{th} processor. Similarly, for all (both factor and variable) vertices $i \in B_k$ we store the associated belief and all inbound messages on the processor k. Each vertex update is therefore a local procedure. For instance, if vertex i is updated, the processor owning vertex i can read factors and all incoming messages without communication. To maintain the locality invariant, updated messages destined to remote vertices are immediately transmitted across the network to the processors owning the destination vertices.

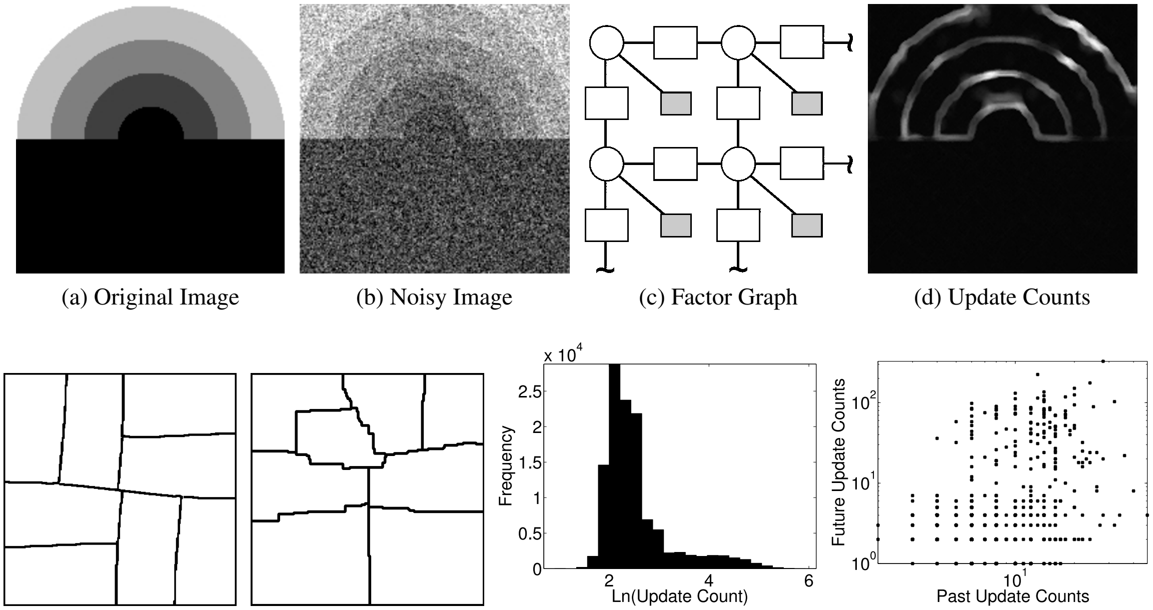
To minimize communication and ensure balanced storage and computation, we frame the minimum communication load balancing objective in terms of a classic balanced graph partitioning:

$$\min_{\mathcal{B}} \quad \sum_{B \in \mathcal{B}} \sum_{(i \in B, j \notin B) \in E} (U_i + U_j) \, |\mathcal{X}_i| \tag{3.21}$$

$$\text{subj. to:} \quad \forall B \in \mathcal{B} \quad \sum_{i \in B} U_i w_i \leq \frac{\gamma}{p} \sum_{v \in v} U_v w_v \tag{3.22}$$

where U_i is the number of times `SendMessages` is invoked on vertex i and w_i is the vertex work defined in Eq. (3.20), and $\gamma \geq 1$ is the balance coefficient. The objective in Eq. (3.21) minimizes communication while the constraint in Eq. (3.22) ensures work balance for small γ and is commonly referred to as the k-way balanced cut objective which is unfortunately NP-Hard in general. However, there are several popular graph partitioning libraries such as METIS [Karypis and Kumar, 1998] and Chaco [Hendrickson and Leland, 1994], which quickly produce reasonable approximations. Here we use the collection of multilevel graph partitioning algorithms in the METIS graph partitioning library. These algorithms, iteratively coarsen the underlying graph, apply high quality partitioning techniques to the small coarsened graph, and then iteratively refine the coarsened graph to obtain a high quality partitioning of the original graph. While there are no theoretical guarantees, these algorithms have been shown to perform well in practice and are commonly used for load balancing in parallel computing.

In the case of static schedules, every vertex is updated the same number of times ($U_i = U_j : \forall i, j,$) and therefore U can be eliminated from both the objective and the constraints. Unfortunately, dynamic algorithms (Wildfire, Residual, and Splash BP) the update counts U_i for each vertex are neither fixed or known. Furthermore, the update counts are difficult to estimate since they depend on the graph structure, factors, and progress towards convergence. Consequently, for dynamic algorithms, we advocate a randomized load balancing technique based on over-partitioning, which does not require knowledge of the number of updates U_i.



(a) Original Image (b) Noisy Image (c) Factor Graph (d) Update Counts

(e) Uninformed Part. (f) Informed Part. (g) Update Distribution (h) Update Prediction

Figure 3.15: We use the synthetic denoising task (also used in Figure 3.13) to illustrate the difficult in estimating the update patterns of Splash belief propagation. **(a)** The original synthetic sunset image which was specifically created to have an irregular update pattern. **(b)** The synthetic image with Gaussian noise added. **(c)** Factor graph model for the true values for the underlying pixels. The latent random variable for each pixel is represented by the circles and the observed values are encoded in the unary factors drawn as shaded rectangles. **(d)** The update frequencies of each variable plotted in log intensity scale with brighter regions updated more frequently. **(e)** Uniformed $U_i = 1$ balanced partitioning. **(f)** The informed partitioning using the true update frequencies after running Splash. Notice that the informed partitioning assigns fewer vertices per processors on the top half of the image to compensate for the greater update frequency. **(g)** The distribution of vertex update counts for an entire execution. **(h)** Update counts from first the half of the execution plotted against update counts from the second half of the execution. This is consistent with phases of execution described in Figure 3.13c.

To illustrate the difficulty involved in estimating the update counts for each vertex, we again use the synthetic denoising task. The input, shown in Figure 3.15a, is a grayscale image with independent Gaussian noise N $\left(0, \sigma^2\right)$ added to each pixel. The factor graph (Figure 3.15c) corresponds to the pairwise grid Markov Random Field constructed by introducing a latent random variable for each pixel and connecting neighboring variables by factors that encode a similarity preference. We also introduce single variable factors which represent the noisy pixel evidence. The synthetic image was constructed to have a nonuniform update pattern (Figure 3.15d) by making the top half more irregular than the bottom half. The distribution of vertex update frequencies (Figure 3.15g) for the denoising task is nonuniform with a few vertices being updated orders of magnitude more frequently than the rest. The update patterns is temporally inconsistent frustrating attempts to estimate future update counts using past behavior (Figure 3.15h).

Uninformed Partitioning

Given the inability to predict the update counts U_i one might conclude, as we did, that sophisticated dynamic graph partitioning is necessary to effectively balance computation and minimize communication. Surprisingly, we find that an uninformed cut obtained by setting the number of updates to a constant (i.e., $U_i = 1$) yields a partitioning with comparable communication cost and work balance as those obtained when using the true update counts. We designate $\hat{\mathcal{B}}$ as the partitioning that optimizes the objectives in Eq. (3.21) and Eq. (3.22) where we have assumed $U_i = 1$. In Table 3.2 we construct uninformed cuts ($p = 120$) for several factor graphs and report the communication cost and balance defined as:

$$\begin{aligned}
\text{Rel. Com. Cost} &= \frac{\sum_{B \in \hat{\mathcal{B}}} \sum_{(u \in B, v \notin B) \in E} w_{uv}}{\sum_{B \in \mathcal{B}^*} \sum_{(u \in B, v \notin B) \in E} w_{uv}} \\
\text{Rel. Work Balance} &= \frac{p}{\sum_{v \in V} w_v} \max_{B \in \hat{\mathcal{B}}} \sum_{v \in B} w_v
\end{aligned}$$

relative to the ideal cut $\mathcal{B}^*$ obtained using the true update counts U_i. We find that uninformed cuts have lower communication costs at the expense of increased imbalance. This discrepancy arises from the need to satisfy the balance requirement with respect to the true U_i at the expense of a higher communication cost.

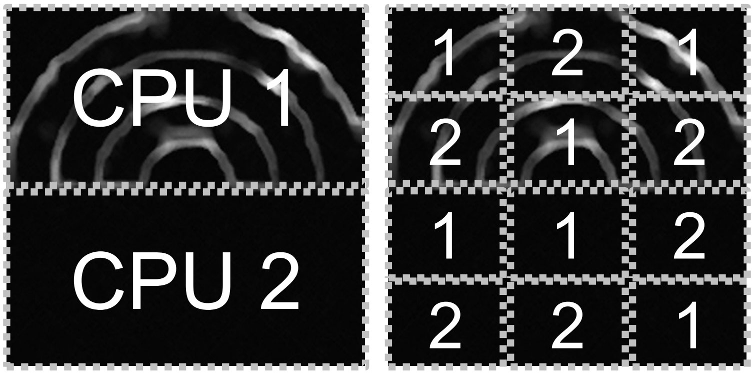
Graph	Rel. Com. Cost	Rel. Work Balance
denoise	0.980	3.44
uw-systems	0.877	1.837
uw-languages	1.114	2.213
cora-1	1.039	1.801

Table 3.2: Comparison of uninformed and informed partitionings with respect to communication cut cost and work balance. While the communication costs of the uniformed partitionings are comparable to those of the informed partitionings, the work imbalance of the uninformed cut is typically greater.

Over-Partitioning

If the graph is partitioned assuming constant update counts, there could be work imbalance if BP is executed using a dynamic schedule. For instance, a frequently updated sub-graph could be placed within a single partition as shown in Figure 3.16a. To decrease the chance of such an event, we can over-partition the graph, as shown in Figure 3.16b, into $k \times p$ balanced partitions and then randomly redistribute the partitions to the original p processors.

By partitioning the graph more finely and randomly assigning regions to different processor, we more evenly distribute nonuniform update patterns improving the overall work balance. However, over-partitioning also increases the number of edges crossing the cut and therefore the communication cost. We demonstrate this effect on the synthetic denoising problem in Figure 3.17a and Figure 3.17b. We also observe that in the limit of $k = n/p$, the over partitioning strategy corresponds exactly to randomly distributing vertices over the processors. This is intuitively, the most balanced partitioning strategy, but also incurs the greatest amount of communication overhead.



(a) Denoise Uniformed Cut (b) Denoise Over-Partitioning

Figure 3.16: Over-partitioning can help improve work balance by more uniformly distributing the graph over the various processors. **(a)** A two processor uninformed partitioning of the denoising factor graph can lead to one processor (CPU1) being assigned most of the work. **(b)** Over-partitioning by a factor of 6 can improve the overall work balance by assigning regions from the top and bottom of the denoising image to both processors.

Choosing the optimal over-partitioning factor k is challenging and depends heavily on hardware, graph structure, and even factors. In situations where the algorithm may be run repeatedly, standard search techniques may be used. We find that in practice a small factor, e.g., $k = 5$ is typically sufficient.

The over-partitioning technique may be justified theoretically. In particular, when using a recursive bisection style partitioning algorithm where the true work split at each step is an unknown random variable, there exists a theoretical bound on the ideal size of k. If at each split the work is divided into two parts of proportion X and $1 - X$ where $\mathbf{E}[X] = \frac{1}{2}$ and $\mathbf{Var}[X] = \sigma^2$ for $\sigma \leq \frac{1}{2}$, Sanders [1994] shows that we can obtain work balance with high probability if we select k to be at least:

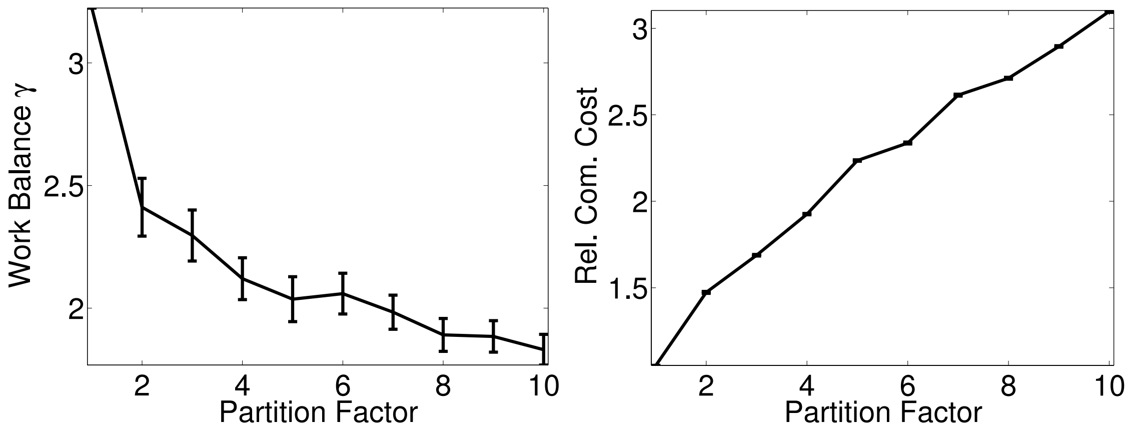
$$\Omega\left(p^{\left(\log\left(\frac{1}{\sigma+1/2}\right)\right)^{-1}}\right).$$

A similar, but looser bound was obtained by Sanders [1996] even if work balance is adversarial.

Incremental Repartitioning

One might consider occasionally repartitioning the factor graph to improve balance. For example, we could divide the execution into T phases and then repartition the graph at the beginning of each phase based on the update patterns from the previous phase. To assess the potential performance gains from incremental repartitioning we conducted a retrospective analysis. We executed the sequential Splash algorithm to convergence on the denoising model and then divided the complete execution into T phases for varying values of T. We then partitioned each phase using the true update counts for that phase and estimate the work imbalance and number of transmitted messages. While the true update counts would not be known in practice, they provide an upper bound on the performance gains of the best possible predictor.

In Figure 3.18 we plot the performance of both optimal phased partitioning and phased partitioning in which future update counts are predicted from previous update counts. In both cases we consider phased



(a) Denoise Work Balance

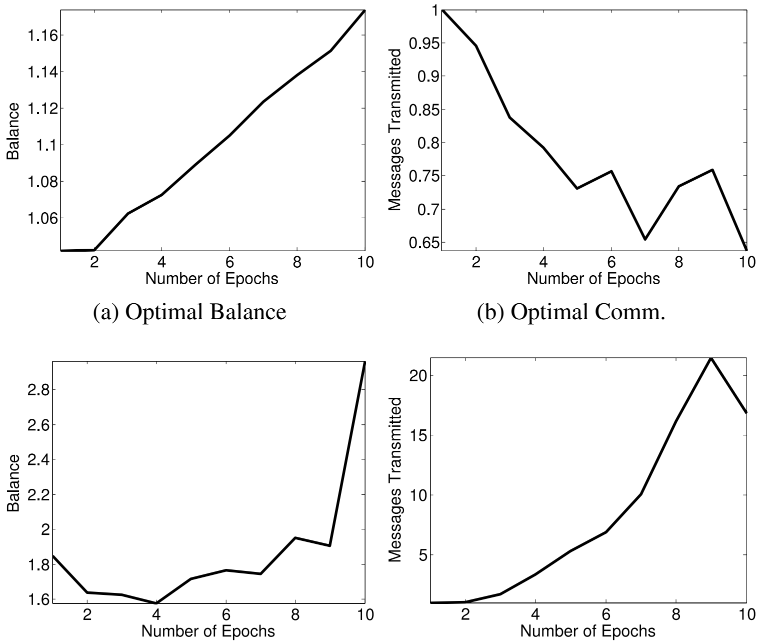
(b) Denoise Rel. Com. Cost

Figure 3.17: The effect of over-partitioning on the work balance and communication cost. For all points 30 trials with different random assignments are used and 95% confidence intervals are plotted. **(a)** The ratio of the size of the partition containing the most work, relative to the ideal size (smaller is better). **(b)** The communication cost relative to the informed partitioning (smaller is better).

partitioning for $T \in \{1, \ldots, 10\}$ phases on a simulated 16 processor system. In Figure 3.18a we see that by repartitioning more often we actually slightly increase the average imbalance over the epochs. We attribute the slight increased imbalance to the increasingly irregular local update counts relative to the connectivity structure (the regular grid). However, as seen in Figure 3.18b the number of messages transmitted over the network (i.e., the communication cost) relative to the optimal single phase partitioning actually decreases by roughly 35%. Alternatively, using the update counts from the previous phase to partition the next phase confers a substantial decrease in overall balance (Figure 3.18c) and a dramatic increase in the communication costs (Figure 3.18d). We therefore conclude that only in situations where update counts can be accurately predicted and network communication is the limiting performance can incremental repartitioning lead to improved performance.

3.8.2 Distributing Scheduling

The SplashBP and ResidualBP algorithms rely on a shared global priority queue. Because a centralized ordering is too costly in the distributed setting, each processor maintains its own local priority queue over its vertices. At any point in the distributed execution one of the processor is always applying the Splash operation to the globally highest residual vertex. Unfortunately, the remaining $p - 1$ highest vertices are not guaranteed to be at the top of the remaining queues and so we do not recover the original shared memory scheduling. However, any processor with vertices that have not yet converged, must eventually update those vertices and therefore can always make progress by updating the vertex at the top of its local queue. In Section 3.8.4 we show that the collection of local queues is sufficient to retain the original optimality properties of the Splash algorithm.



(a) Optimal Balance

(b) Optimal Comm.

(c) Naive Balance

(d) Naive Comm.

Figure 3.18: Here we plot the balance and communication costs of incremental repartitioning against the number of epochs (repartitioning phases) in a single execution. The balance is measured as the work ratio between the smallest and largest work block. We present the messages transmitted measured relative to the optimal performance for a single phase rather than raw message counts. **(a,b)** The balance and communication costs for optimal phased partitioning where the update counts in each phase are known in advance. **(c,d)** The balance and communication costs for naive phased partitioning where the update counts in each phase are assumed to be the update counts observed in the previous phase.

3.8.3 Distributed Termination

In the absence of a common synchronized state it is difficult to assess whether a convergence condition has been globally achieved. The task of assessing convergence is complicated by the fact that when a node locally converges, it could be *woken up* at any time by an incoming message which may causes its top residual to exceed the termination threshold.

Fortunately, the *distributed termination problem* [Matocha and Camp, 1998, Mattern, 1987] is well studied with several solutions. We implement a variation of the algorithm described in Misra [1983]. The algorithm (Alg. 3.14) pases a token comprising of 2 integer values, `NumSent` and `NumReceived`, in a fixed loop over the machines. The integer values represent the total number of network messages sent and received throughout the execution of the program across all machines. When the node holding onto the token converges, the node will add the number of network messages it has sent and received since the last time it has seen the token, to `NumSent` and `NumReceived` respectively. The node will then forward the token to the next node in the ring. Global termination is achieved when the token completes a loop around the ring without changing, and `NumSent` equals `NumReceived`.

Algorithm 3.14: Distributed Termination Algorithm

```
Output: Converged
Data: Global Variables: LastToken, NetMessagesSent, NetMessagesReceived
// Infinite Loop
while True do
    if I have the Token then
        // Check for the global termination
        if Token == LastToken AND Token.NumSent == Token.NumReceived then
            return Converged = True
        else
            // Update the Token with this node's contributions
            // Reset the network message counters
            Token.NumSent ← Token.NumSent + NetMessagesSent
            Token.NumReceived ← Token.NumReceived + NetMessagesReceived
            NetMessagesSent ← 0
            NetMessagesReceived ← 0
            LastToken ← Token // Remember the last Token we saw
            Transmit(Token, NextNode) // Send the updated token to the next
                node in the ring

    Sleep and Wait For Any Incoming Network Message.

    if Network Message is not a Token then
        // This is not a message.  Wake up and process it
        return Converged = False
```

3.8.4 The Distributed Splash Algorithm

We now present the distributed Splash algorithm (Alg. 3.15) which can be divided into two phases, setup and inference. In the setup phase, in Line 1 we over-segment the input factor graph into kp pieces using the METIS partitioning algorithm. Note that this could be accomplished in parallel using ParMETIS, however our implementation uses the sequential version for simplicity. Then in Line 2 we randomly assign k pieces to each of the p processors. In parallel each processor collects its factors and variables (Line 3). On Line 4 the priorities of each variable and factor are set to infinity to ensure that every vertex is updated at least once.

On Line 5 we evaluate the top residual with respect to the β convergence criterion and check for termination in the token ring. On Line 6, the `DynamicSplash` operation is applied to v. In the distributed setting Splash construction procedure does not cross partitions. Therefore each Splash is confined to the vertices on that processor. After completing the Splash all external messages from other processors are incorporated (Line 7). Any beliefs that changed during the Splash or after receiving external messages are promoted in the priority queue on Line 9. On Line 10, the external messages are transmitted across the network. The process repeats until termination at which point all beliefs are sent to the originating processor.

Empirically, we find that accumulating external messages and transmitting only once every 5-10 loops

Algorithm 3.15: The Distributed Splash Algorithm

```
// The graph partitioning phase =========================>
// Construct factor $k$ over-partitioning for $p$ processors
$\mathcal{B}_{\text{temp}} \leftarrow$ OverSegment($G, p, k$);
// Randomly assign over-partitioning to the original processors
$\mathcal{B} \leftarrow$ RandomAssign($\mathcal{B}_{temp}, p$);
// The distributed inference phase ======================>
forall $b \in \mathcal{B}$ do in parallel
    // Collect the variables and factors associated with this
       partition
    Collect($\mathcal{F}_b, \mathbf{x}_b$) ;
    // Initialize the local priority queue
    Initialize($Q$);
    // Loop until TokenRing signifies convergence
    while TokenRing($Q, \beta$) do
        $v \leftarrow$ Top($Q$) ;
        DynamicSplash($v$, $W_{\max}$, $\beta$);
        RecvExternalMsgs();
        // Update priorities affected by the Splash and newly
           received messages
        foreach $u \in$ local changed vertices do
            Promote($Q$, $||\Delta b_v||_1$);
        SendExternalMsgs();
```

tends to increase performance by substantially decreasing network overhead. Accumulating messages may however adversely affect convergence on some graphical models. To ensure convergence in our experiments, we transmit on every iteration of the loop.

We now show that Splash retains the optimality in the distributed setting.

Theorem 3.8.1 (Splash Chain Optimality)**.** *Given a chain graph with $n = n$ vertices and $p \leq n$ processes, the distributed Splash algorithm with no over-segmentation, using a graph partitioning algorithm which returns connected partitions, and with work Splash size at least $2 \sum_{v \in V} w_v / p$ will obtain a τ_ϵ-approximation in expected running time $O\left(\frac{n}{p} + \tau_\epsilon\right)$.*

Proof of Theorem 3.8.1. We assume that the chain graph is optimally sliced into p connected pieces of n/p vertices each. Since every vertex has at most 2 neighbors, the partition has at most $2n/p$ work. A Splash anywhere within each partition will therefore cover the entire partition, performing the complete *forward-backward* update schedule.

Because we send and receive all external messages after every splash, after $\lceil \frac{\tau_\epsilon}{n/p} \rceil$ iterations, every vertex will have received messages from vertices a distance of at least τ_ϵ away. The runtime will therefore be:

$$\frac{2n}{p} \times \left\lceil \frac{\tau_\epsilon}{n/p} \right\rceil \leq \frac{2n}{p} + 2\tau_\epsilon$$

Since each processor only send 2 external messages per iteration (one from each end of the partition), communication therefore only adds a constant to the total runtime. □

3.8.5 Algorithm Comparison in the Distributed Setting

Distributed experiments were conducted on cluster composed of 13 blades (nodes) each with 2 Intel Xeon E5345 2.33GHz Quad core processors. Each processor is connected to the memory on each node by a single memory controller. The nodes are connected via a fast Gigabit ethernet switch. We invoked the weighted *kmetis* partitioning routine from the METIS software library for graph partitioning and partitions were computed in under 10 seconds.

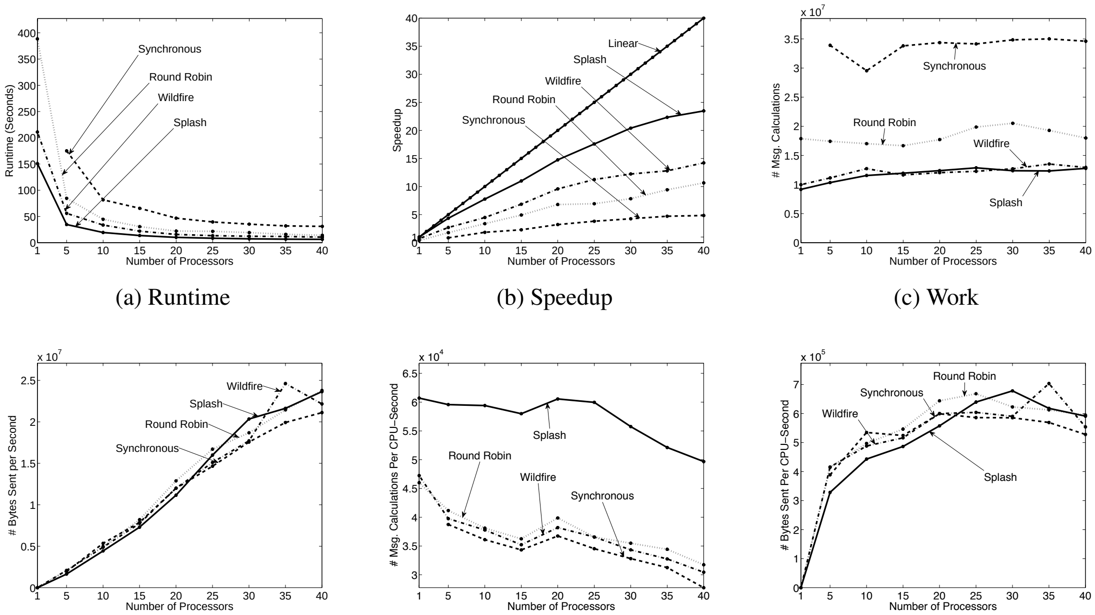
To assess the performance of the BP algorithms in the distributed setting we focus our attention on larger models. To adequately challenge the distributed algorithms we consider the Elidan protein networks (with a stricter termination bound $\beta = 10^{-10}$) and the larger UW-CSE MLNs with the termination bound $\beta = 10^{-5}$. All the baseline algorithms converged on the Elidan protein networks except synchronous running on one processor which ran beyond the 10,000 second cutoff. Consistent with the multicore results, only the Splash algorithm converges on the larger UW-CSE MLNs.

Since all the baseline algorithms converged on the smaller Elidan networks we use these networks for algorithm comparison. Because the single processor runtime on these networks is only a few minutes we analyze performance from 1 to 40 processors in increments of 5 rather than using the entire 104 processor system. As discussed in the experimental setup, we consider over-partition factors of 2, 5, and 10 for each algorithm and present the results using the best over-partition factors for each of the respective algorithms (which is typically 5). In Figure 3.19a and Figure 3.19b we present the standard runtime and speedup curves which show that the Splash algorithm achieves the best runtime and speedup performance with a maximum speedup of roughly 23 at 40 processors. While the speedup does not scale linearly we achieve a runtime of 6.4 seconds at 40 processors making it difficult to justify extending to an additional 80 processors. We also plot the amount of work Figure 3.19c as a function of the number of processors and find that despite the message delays and distributed scheduling, the algorithmic efficiency remains relatively constant with the Splash and Wildfire algorithm performing optimally.

In Figure 3.19d we plot the total amount of network traffic measured in bytes as a function of the number of processors and we find that the Splash algorithm performs the minimal amount of network communication. Furthermore the amount of network communication scales linearly with the number of processors. We also find in Figure 3.19e that the Splash algorithm is able to more fully utilize the cluster by executing more message calculations per processor-second and doing so consistently as the number of processor scales. Finally, we compare the algorithms in terms of their communication efficiency measured in total bytes sent per processor-second. The total bytes sent per processor-second, plotted in Figure 3.19f is a measure of the network bandwidth used per machine. Again we find that the Splash requires the minimal network bandwidth and that the network bandwidth grows sub-linearly with the number of processors. At peak load we use less than a megabyte per second per machine.

The larger UW-CSE MLNs provide the best demonstration of the effectiveness of the Splash algorithm. None of the other inference algorithms converge on these models, and the models are sufficiently large to justify the use of a cluster. In Figure 3.20 and Figure 3.21, we provide plots for the UW-Systems and the UW-AI MLNS respectively.

The UW-Systems model is larger, and we could demonstrate a linear to super-linear speedup up to about 65



(a) Runtime

(b) Speedup

(c) Work

(d) Communication

(e) Work Efficiency

(f) Comm. Efficiency

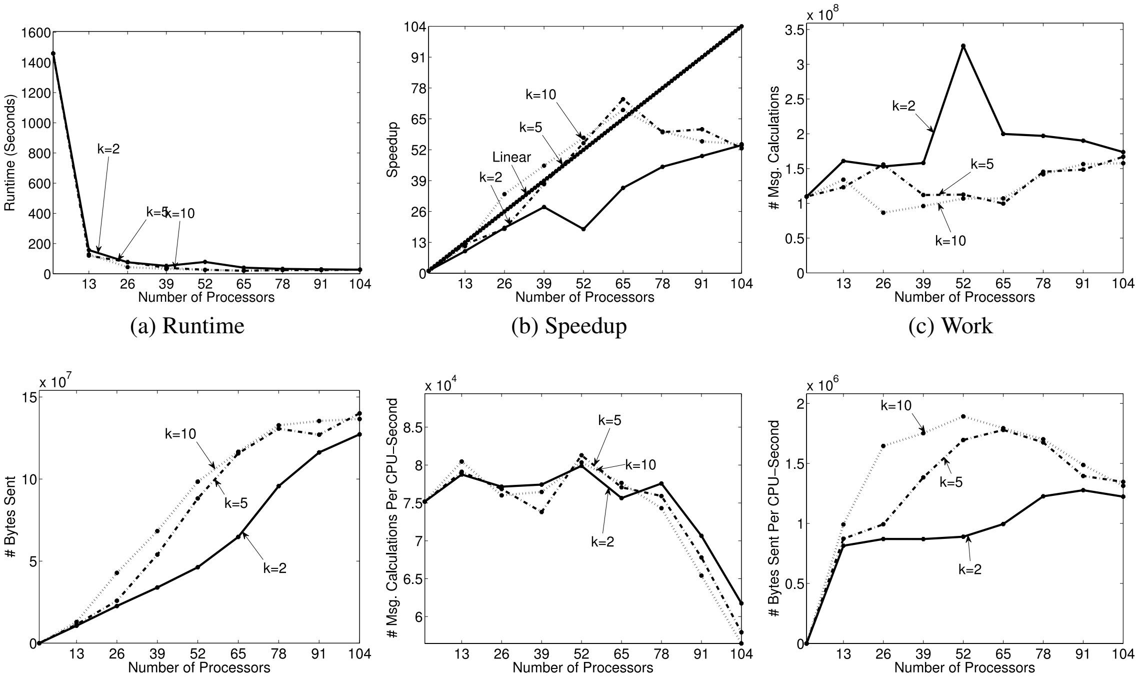
Figure 3.19: Distributed Results for the Elidan1 Protein Network

processors. At 65 processors, the runtime is about 23.6 seconds with a partition factor of 10. The speedup curves start to flatten out beyond that. The plot in Figure 3.20b justify the claim that over-partitioning can increase performance through improved load-balancing. Over-partitioning however, does lead to increased network communication as seen in Figure 3.20d and Figure 3.20f. At 65 processors, over 100 MB of network messages were communicated in 23 seconds.

The UW-AI model in Figure 3.21 has similar results. However, as the model is smaller, we notice that the speedup curves start to flatten out earlier, demonstrating linear speedup only up to about 39 processors. Once again, at 65 processors, the runtime is only 22.7 seconds with a partition factor of 10.

3.9 Additional Related Work

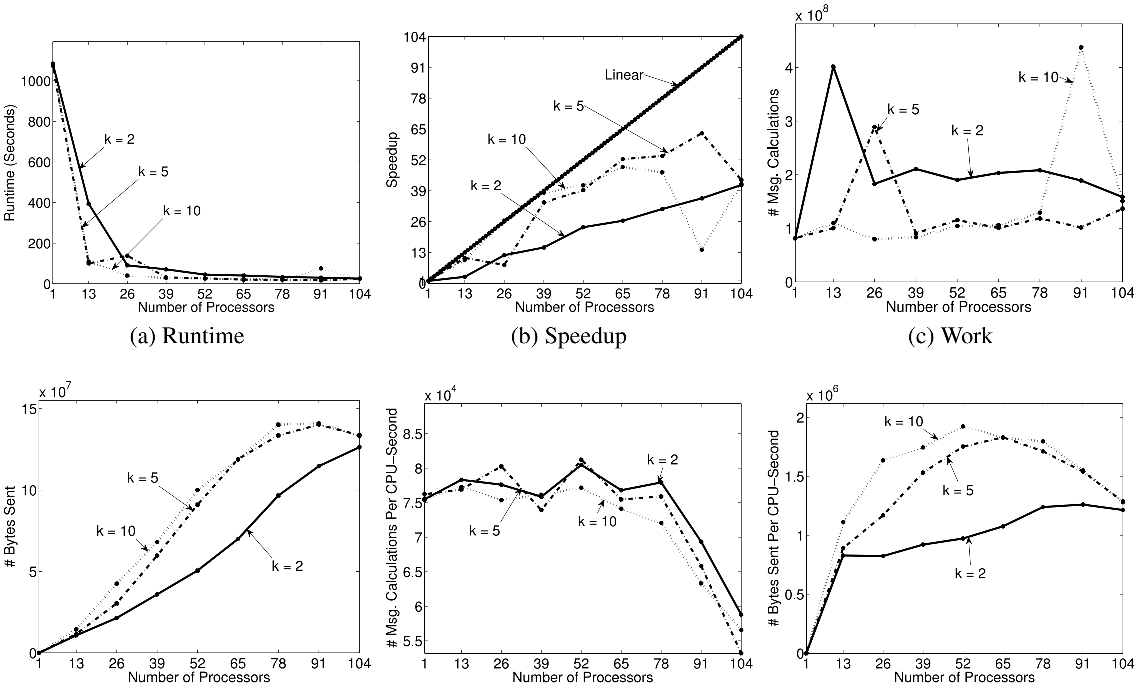
While we focus on approximate inference methods, much of the existing work on parallel algorithms for graphical models focuses on exact inference which is extremely computationally intensive. Most exact inference algorithms operate on the junction tree which is an acyclic representation of the factorized distribution. One of the earliest parallel exact inference algorithm was introduced by Kozlov and Singh [1994] who implemented the Lauritzen-Spiegelhalter (see Koller and Friedman [2009]) algorithm on a 32 core Stanford DASH processor. Later Pennock [1998] provided a method to both construct the junction tree and apply Lauritzen-Spiegelhalter with $\log$ depth inference using a form of parallel tree contraction [Miller and Reif, 1985]. While the work by Pennock [1998] provides an optimal algorithm it requires substantial fine grain coordination. Later Xia and Prasanna [2008, 2010] and Xia et al. [2009] implemented the algorithm for various platforms and demonstrated limited scaling on models with high treewidth. Unfortunately, the complexity of inference in junction trees is exponential in the treewidth which grows quickly in loopy models with many variables. Therefore, for most real-world large-scale models the treewidth is typically prohibitively large.



(a) Runtime (b) Speedup (c) Work

(d) Communication (e) Comp. Efficiency (f) Com. Efficiency

Figure 3.20: Distributed Parallel Results for the UW-Systems MLN



(a) Runtime (b) Speedup (c) Work

(d) Communication (e) Comp. Efficiency (f) Com. Efficiency

Figure 3.21: Distributed Parallel Results for the UW-AI MLN

In the sequential setting their has been an increased emphasis on asynchronous prioritized scheduling for belief propagation. Soni et al. [2010] examined the use of asynchronous *domain specific* scheduling to accelerate convergence. Prior to our work on SplashBP, Wainwright et al. [2006] proposed an alternative factorization of approximate inference algorithms along spanning trees. This work conceptually corresponds to scheduling message updates along spanning trees. After our work on SplashBP, Tarlow et al. [2011] proposed a prioritized tree based inference algorithm for max-product linear program solver which closely resembles SplashBP. It is encouraging to see that the GrAD methodology not only leads to efficient parallel algorithms but is equally applicable in the design of efficient sequential algorithms.

The τ_ϵ-approximation can be related to work in approximate message passing algorithms. In particular, work by Ihler et al. [2005] studied the effect of errors in the convergence of belief propagation. They related the *dynamic range* of factors, a measure of sensitivity to changes in message values, to the convergence and accuracy of loopy belief propagation. Consistent with our τ_ϵ-approximation they found that messages errors typically decay over long chains.

3.10 Conclusion

Approximate parallel inference algorithms are important to a wide range of machine learning applications. In this chapter we reviewed the natural parallelization of the belief propagation algorithm using the synchronous schedule and demonstrated both theoretically and empirically, how this can lead to a highly inefficient parallel algorithm.

Using the GrAD design principles we gradually improved upon the Synchronous BP algorithm by gradually introducing asynchronous dynamic scheduling. This lead to the sequence of algorithms, Round Robin, Wildfire, and Residual BP, each improving upon the previous. Finally by identifying the serial sub-problems, chains and more generally trees, we developed the Splash BP algorithm, which sequentially moves messages along spanning trees. We theoretically demonstrated that the SplashBP algorithm is optimal on chains but generalizes to arbitrary cyclic models.

We discussed how to implement each parallel belief propagation algorithm in the shared memory setting using basic locking primitives and parallel priority queues. We identified load balancing and communication contention in the context of dynamic scheduling as key challenges to a distributed belief propagation algorithms and presented a simple over-partitioning method to address these challenges. Finally, we experimentally compared each parallel belief propagation algorithm demonstrating the substantial gains of applying the GrAD design methodology in both the shared memory and distributed memory settings.

Chapter 4

Probabilistic Inference: Gibbs Sampling

Gibbs sampling is a popular MCMC inference procedure used widely in statistics and machine learning. However, on many models the Gibbs sampler can be slow mixing [Barbu and Zhu, 2005, Kuss and Rasmussen, 2005]. Consequently, a number of authors [Asuncion et al., 2008, Doshi-Velez et al., 2009, Newman et al., 2007, Yan et al., 2009] have proposed parallel methods to accelerate Gibbs sampling. Unfortunately, most of the recent methods rely on *synchronous* Gibbs updates that are not ergodic, and therefore generate chains that do not converge to the targeted stationary distribution.

By applying the GrAD methodology we introduce three ergodic parallel Gibbs samplers. The first, called the **Chromatic** sampler, applies a classic technique relating graph coloring to parallel job scheduling, to obtain a direct parallelization of the classic sequential scan Gibbs sampler. We show that the Chromatic sampler is provably ergodic and provide strong guarantees on the parallel reduction in mixing time. We then introduce the **Asynchronous Sampler** which relaxes the Chromatic schedule and introduces the Markov Blanket Locking protocol to allow greater asynchrony while preserving ergodicity.

For the relatively common case of models with two-colorable Markov random fields, the Chromatic sampler provides substantial insight into the behavior of the non-ergodic Synchronous Gibbs sampler. We show that in the two-colorable case, the Synchronous Gibbs sampler is equivalent to the simultaneous execution of two independent Chromatic samplers and provide a method to recover the corresponding ergodic chains. As a consequence, we are able to derive the invariant distribution of the Synchronous Gibbs sampler and show that it is ergodic with respect to functions over single variable marginals.

The Chromatic sampler achieves a linear increase in the rate at which samples are generated and is therefore ideal for models where the variables are weakly coupled. However, for models with strongly coupled variables, the chain can still mix prohibitively slowly. In this case, it is often necessary to jointly sample large subsets of related random variables [Barbu and Zhu, 2005, Jensen and Kong, 1996] in what is known as a blocking Gibbs sampler.

The third parallel Gibbs sampler, the **Splash** sampler, addresses the challenges of highly correlated variables by incrementally constructing multiple bounded tree-width blocks, called Splashes, and then jointly sampling each Splash using parallel junction-tree inference and backward-sampling. To ensure that multiple simultaneous Splashes are conditionally independent (and hence that the chain is ergodic), we introduce a Markov blanket locking protocol. To accelerate burn-in and ensure high likelihood states are reached quickly, we introduce a vanishing *adaptation* heuristic for the initial samples of the chain, which explicitly builds blocks of strongly coupled variables.

We construct optimized implementation of the Chromatic and Splash parallel samplers and compare performance on synthetic and real-world sampling problems using a 32 processor multicore system. The Asynchronous sampler is a special case of the Splash sampler with Splash size set to one. We find that both algorithms achieve strong speedups in sample generation, and the adaptive Splash sampler can further accelerate mixing on strongly correlated models. Our experiments illustrate that the two sampling strategies complement each other: for weakly coupled variables, the Chromatic sampler performs best, whereas the Splash sampler is needed when strong dependencies are present.

4.1 The Gibbs Sampler

The Gibbs sampler was introduced by Geman and Geman [1984] and is a popular Markov Chain Monte Carlo (MCMC) algorithm used to simulate samples from the joint distribution. The Gibbs sampler is constructed by iteratively sampling each variable,

$$X_i \sim \mathbf{P}\left(X_i \,|\, \mathbf{X}_{\mathcal{N}_i} = \mathbf{x}_{\mathcal{N}_i}\right) \propto \prod_{\mathbf{A}: i \in \mathbf{A}, \mathbf{A} \in \mathcal{F}} f_{\mathbf{A}}(X_i, \mathbf{x}_{\mathcal{N}_i}) \tag{4.1}$$

given the assignment to the variables in its Markov blanket. Geman and Geman [1984] showed that if each variable is sampled infinitely often and under reasonable assumptions on the conditional distributions (e.g., positive support), the Gibbs sampler is ergodic (i.e., it converges to the true distribution). While we have considerable latitude in the update schedule, we shall see in subsequent sections that certain updates must be treated with care: in particular, Geman and Geman [1984] were incorrect in their claim that parallel simultaneous sampling of all variables (the *Synchronous* update) yields an ergodic chain.

For large models with complex dependencies, the mixing time and even the time required to obtain a high likelihood sample can be substantial. Therefore, we would like to use parallel resources to increase the speed of the Gibbs sampler. The simplest method to construct a parallel sampler is to run a separate chain on each processor. However, running multiple parallel chains requires large amounts of memory and, more importantly, is not guaranteed to accelerate mixing or the production of high-likelihood samples. As a consequence, we focus on single chain parallel acceleration, where we apply parallel methods to increase the speed at which a single Markov chain is advanced. The single chain setting ensures that any parallel speedup directly contributes to an equivalent reduction in the mixing time, and the time to obtain a high-likelihood sample.

Unfortunately, recent efforts to build parallel single-chain Gibbs samplers have struggled to retain ergodicity. The resulting methods have relied on approximate sampling algorithms [Asuncion et al., 2008] or proposed generally costly extensions to recover ergodicity [Doshi-Velez et al., 2009, Ferrari et al., 1993, Newman et al., 2007]. Here we will apply the GrAD methodology to design efficient parallel Gibbs samplers while ensuring ergodicity.

4.2 The Synchronous Gibbs Sampler

A naive single chain parallel Gibbs sampler is obtained by sampling all variables simultaneously on separate processors. Called the **Synchronous** Gibbs sampler, this highly parallel algorithm (Alg. 4.1) was originally

Algorithm 4.1: The Synchronous Gibbs Sampler

```
forall Variables $X_j$ do in parallel
    Execute Gibbs Update: $X_j^{(t+1)} \sim \mathbf{P}\left(X_j \mid \mathbf{x}_{\mathcal{N}_j}^{(t)}\right)$
end
```

proposed by Geman and Geman [1984]. Unfortunately the extreme parallelism of the Synchronous Gibbs sampler comes at a cost. As others (e.g., Newman et al. [2007]) have observed, one can easily construct cases where the Synchronous Gibbs sampler is not ergodic and therefore does not converge to the correct stationary distribution.

For example, suppose we draw $\left(X_1^{(t-1)}, X_2^{(t-1)}\right)$ from the multivariate normal distribution:

$$\begin{pmatrix} X_1^{(t-1)} \\ X_2^{(t-1)} \end{pmatrix} \sim \mathrm{N}\left[\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} \sigma_1^2 & \rho \\ \rho & \sigma_2^2 \end{pmatrix}\right]$$

and then use the Synchronous Gibbs sampler to generate $\left(X_1^{(t)}, X_2^{(t)}\right)$ given $\left(X_1^{(t-1)}, X_2^{(t-1)}\right)$. One can easily show that:

$$\begin{pmatrix} X_1^{(t)} \\ X_2^{(t)} \end{pmatrix} \sim \mathrm{N}\left[\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} \sigma_1^2 & \frac{\rho^3}{\sigma_1^2\sigma_2^2} \\ \frac{\rho^3}{\sigma_1^2\sigma_2^2} & \sigma_2^2 \end{pmatrix}\right]$$

where the covariance term is not preserved. Note, however that the marginals remain consistent. Alternatively, in the discrete case, consider the two binary variable discrete model $\mathbf{P}(X_1, X_2)$ where

$$\mathbf{P}(X_1 = x_1, X_2 = x_2) \propto \begin{cases} \epsilon & x_1 \neq x_2 \\ 1-\epsilon & x_1 = x_2 \end{cases} \tag{4.2}$$

for some $0 < \epsilon < 1$. The transition matrix of the Synchronous sampler which samples from both variables simultaneously is therefore:

	(0,0)	(0,1)	(1,0)	(1,1)
(0,0)	$(1-\epsilon)^2$	$(1-\epsilon)\epsilon$	$(1-\epsilon)\epsilon$	ϵ^2
(0,1)	$(1-\epsilon)\epsilon$	ϵ^2	$(1-\epsilon)^2$	$(1-\epsilon)\epsilon$
(1,0)	$(1-\epsilon)\epsilon$	$(1-\epsilon)^2$	ϵ^2	$(1-\epsilon)\epsilon$
(1,1)	ϵ^2	$(1-\epsilon)\epsilon$	$(1-\epsilon)\epsilon$	$(1-\epsilon)^2$

which has the incorrect stationary distribution (first eigenvector is 1):

$$\tilde{\mathbf{P}}(X_1, X_2) \propto 1$$

Again, the marginals $\tilde{\mathbf{P}}(X_1) = \mathbf{P}(X_1)$ and $\tilde{\mathbf{P}}(X_2) = \mathbf{P}(X_2)$ are correctly estimated.

Not only does the Synchronous Gibbs sampler converge to the wrong distribution, it can actually decelerate burn-in, the time it takes to reach high-likelihood samples. If we return to the simple two variable discrete model and start with an assignment where $X_1 \neq X_2$ then the probability that $X_1 = X_2$ after a single iteration is:

$$\mathbf{P}\left(X_1^{\text{new}} = X_2^{\text{new}} \mid X_1^{\text{old}} \neq X_2^{\text{old}}\right) = 2\epsilon(1-\epsilon) \leq 2\epsilon$$

Therefore for small ϵ the probability of transitioning to a high likelihood state vanishes when run synchronously but occurs almost surely when run *asynchronously*.

Algorithm 4.2: The Chromatic Sampler

```
Input: $k$-Colored MRF
for For each of the $k$ colors $\kappa_i : i \in \{1, \ldots, k\}$ do
    forall Variables $X_j \in \kappa_i$ in the $i^{th}$ color do in parallel
        Execute Gibbs Update: $X_j^{(t+1)} \sim \mathbf{P}\left(X_j \mid \mathbf{x}^{(t+1)}_{\mathcal{N}_j \in \kappa_{<i}}, \mathbf{x}^{(t)}_{\mathcal{N}_j \in \kappa_{>i}}\right)$
    end
end
```

4.3 The Chromatic Sampler

Just as with the Synchronous BP algorithm in Chapter 3, we can improve upon the Synchronous Gibbs sampler by scheduling computation asynchronously. In particular we want to construct a parallel execution of Gibbs updates that has an equivalent sequential execution and therefore retains ergodicity. We can achieve this serial equivalence by applying a classic technique from parallel computing and introduce the Chromatic parallel Gibbs sampler.

The Chromatic Gibbs sampler (Alg. 4.2) begins by constructing a k-coloring of the MRF. A k-coloring of the MRF assigns one of k colors to each vertex such that adjacent vertices have different colors. The Chromatic sampler simultaneously draws new values for all variables in one color before proceeding to the next color. The k-coloring of the MRF ensures that all variables within a color are conditionally independent given the variables in the remaining colors and can therefore be sampled independently and in parallel.

By combining a classic result ([Bertsekas and Tsitsiklis, 1989, Proposition 2.6]) from parallel computing with the original Geman and Geman [1984] proof of ergodicity for the sequential Gibbs sampler one can easily show:

Proposition 4.3.1 (Graph Coloring and Parallel Execution). *Given p processors and a k-coloring of an n-variable MRF, the parallel Chromatic sampler is ergodic and generates a new joint sample in running time:*

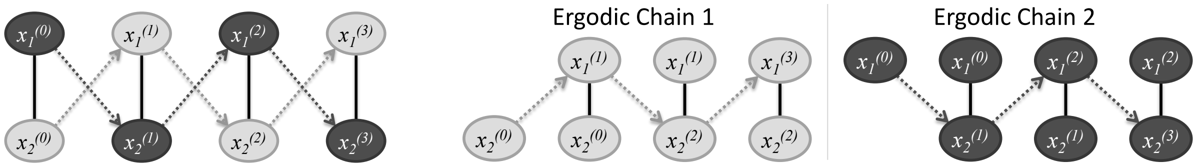
$$O\left(\frac{n}{p} + k\right).$$

Proof. From [Bertsekas and Tsitsiklis, 1989, Proposition 2.6] we know that the parallel execution of the Chromatic sampler corresponds exactly to the execution of a sequential scan Gibbs sampler for some permutation over the variables. The running time can be easily derived:

$$O\left(\sum_{i=1}^{k} \left\lceil \frac{|\kappa_i|}{p} \right\rceil\right) = O\left(\sum_{i=1}^{k} \left(\frac{|\kappa_i|}{p} + 1\right)\right) = O\left(\frac{n}{p} + k\right).$$

□

Therefore, given sufficient parallel resource ($p \in O(n)$) and a k-coloring of the MRF, the parallel Chromatic sampler has running-time $O(k)$, which for many modeling tasks is constant in the number of vertices. It is important to note that this parallel gain directly results in a factor of p reduction in the mixing time as well as the time to reach a high-likelihood sample.



(a) Synchronous Chain

(b) Two Ergodic Chains

Figure 4.1: **(a)** Execution of a two colored model using the synchronous Gibbs sampler. The dotted lines represent dependencies between samples. **(b)** Two ergodic chains obtained by executing the Synchronous Gibbs sampler. Note that ergodic sums with respect to marginals are equivalent to those obtained using the Synchronous sampler.

Unfortunately, constructing the minimal coloring of a general MRF is NP-Complete. However, for many common models the optimal coloring can be quickly derived. For example, given a plate diagram, we can typically compute an optimal coloring of the plates, which can then be applied to the complete model. When an optimal coloring cannot be trivially derived, we find simple graph coloring heuristics (see Kubale [2004]) perform well in practice.

4.3.1 Properties of 2-Colorable Models

Many common models in machine learning have natural two-colorings. For example, Latent Dirichlet Allocation, the Indian Buffet process, the Boltzmann machine, hidden Markov models, and the grid models commonly used in computer vision all have two-colorings. For these models, the Chromatic sampler provides additional insight into properties of the Synchronous sampler. The following theorem relates the Synchronous Gibbs sampler to the Chromatic sampler in the two-colorable setting and provides a method to recover two ergodic chains from a single Synchronous Gibbs chain:

Theorem 4.3.2 (2-Color Ergodic Synchronous Samples). *Let $(X^{(t)})_{t=0}^m$ be the non-ergodic Markov chain constructed by the Synchronous Gibbs sampler (Alg. 4.1) then using only $(X^{(t)})_{t=0}^m$ we can construct two ergodic chains $(Y^{(t)})_{t=0}^m$ and $(Z^{(t)})_{t=0}^m$ which are conditionally independent given $X^{(0)}$ and correspond to the simultaneous execution of two Chromatic samplers (Alg. 4.2).*

Proof. We split the chain $(X^{(t)})_{t=0}^m = (X_{\kappa_1}^{(t)}, X_{\kappa_2}^{(t)})_{t=0}^m$ over the two colors and then construct the chains $(Y^{(t)})_{t=0}^m$ and $(Z^{(t)})_{t=0}^m$ by simulating the two Chromatic Gibbs samplers, which each advance only one color at a time conditioned on the other color (as illustrated in Figure 4.1):

$$\begin{aligned}\left(Y^{(t)}\right)_{t=0}^m &= \left[\begin{pmatrix} X_{\kappa_1}^{(0)} \\ X_{\kappa_2}^{(1)} \end{pmatrix}, \begin{pmatrix} X_{\kappa_1}^{(2)} \\ X_{\kappa_2}^{(1)} \end{pmatrix}, \begin{pmatrix} X_{\kappa_1}^{(2)} \\ X_{\kappa_2}^{(3)} \end{pmatrix}, \ldots\right] \\ \left(Z^{(t)}\right)_{t=0}^m &= \left[\begin{pmatrix} X_{\kappa_1}^{(1)} \\ X_{\kappa_2}^{(0)} \end{pmatrix}, \begin{pmatrix} X_{\kappa_1}^{(1)} \\ X_{\kappa_2}^{(2)} \end{pmatrix}, \begin{pmatrix} X_{\kappa_1}^{(3)} \\ X_{\kappa_2}^{(2)} \end{pmatrix}, \ldots\right]\end{aligned}$$

Observe that no samples are shared between chains, and given $X^{(0)}$ both chains are independent. Finally, because both derived chains are simulated from the Chromatic sampler they are provably ergodic. □

Using the partitioning induced by the 2-coloring of the MRF we can analytically construct the invariant distribution of the Synchronous Gibbs sampler:

Theorem 4.3.3 (**Invariant Distribution of Sync. Gibbs**). *Let $(X_{\kappa_1}, X_{\kappa_2}) = X$ be the partitioning of the variables over the two colors, then the invariant distribution of the Synchronous Gibbs sampler is the product of the marginals $\mathbf{P}(X_{\kappa_1})\,\mathbf{P}(X_{\kappa_2})$.*

Proof. Let the current state of the sampler be $X = (X_{\kappa_1}, X_{\kappa_2})$ and the result of a single Synchronous Gibbs update be $X' = (X'_{\kappa_1}, X'_{\kappa_2})$. By definition the Synchronous Gibbs update simulates $X'_{\kappa_1} \sim \mathbf{P}\left(x'_{\kappa_1} \,|\, x_{\kappa_2}\right)$ and $X'_{\kappa_2} \sim \mathbf{P}\left(x'_{\kappa_2} \,|\, x_{\kappa_1}\right)$. Therefore the transition kernel for the Synchronous Gibbs sampler is:

$$K(x' \,|\, x) = \mathbf{P}\left(x'_{\kappa_1} \,|\, x_{\kappa_2}\right) \mathbf{P}\left(x'_{\kappa_2} \,|\, x_{\kappa_1}\right)$$

We can easily show that $\mathbf{P}(X_{\kappa_1})\,\mathbf{P}(X_{\kappa_2})$ is the invariant distribution of the Synchronous Gibbs sampler:

$$\begin{aligned}
\mathbf{P}\left(x'\right) &= \sum_{x} K(x' \,|\, x) \mathbf{P}(x) \\
&= \sum_{x_{\kappa_1}} \sum_{x_{\kappa_2}} \mathbf{P}\left(x'_{\kappa_1} \,|\, x_{\kappa_2}\right) \mathbf{P}\left(x'_{\kappa_2} \,|\, x_{\kappa_1}\right) \mathbf{P}(x_{\kappa_1}) \mathbf{P}(x_{\kappa_2}) \\
&= \sum_{x_{\kappa_1}} \sum_{x_{\kappa_2}} \mathbf{P}\left(x_{\kappa_1}, x'_{\kappa_2}\right) \mathbf{P}\left(x'_{\kappa_1}, x_{\kappa_2}\right) \\
&= \mathbf{P}\left(x'_{\kappa_1}\right) \mathbf{P}\left(x'_{\kappa_2}\right)
\end{aligned}$$

□

A useful consequence of Theorem 4.3.3 is that when computing ergodic averages over sets of variables with the same color, we can directly use the non-ergodic Synchronous samples and still obtain convergent estimators:

Corollary 4.3.4 (**Monochromatic Marginal Ergodicity**). *Given a sequence of samples $(x^{(t)})_{t=0}^{m}$ drawn from the Synchronous Gibbs sampler on a two-colorable model, empirical expectations computed with respect to single color marginals are ergodic:*

$$\forall f, i \in 1, 2 : \lim_{m \to \infty} \frac{1}{m} \sum_{t=0}^{m} f(x_{\kappa_i}^{(t)}) \xrightarrow{a.s.} \mathbf{E}\left[f(X_{\kappa_i})\right]$$

Corollary 4.3.4 justifies many applications where the Synchronous Gibbs sampler is used to estimate single variables marginals and explains why the Synchronous Gibbs sampler performs well in these settings. However, Corollary 4.3.4 also highlights the danger of computing empirical expectations over variables that span both colors without splitting the chains as shown in Theorem 4.3.2.

We have shown that both the Chromatic sampler and the Synchronous sampler can provide ergodic samples. However the Chromatic sampler is clearly superior when the number of processors is less than the half the number of vertices ($p < n/2$) since it will advance a single chain twice as fast as the Synchronous sampler.

4.4 Asynchronous Gibbs Sampler

While the Chromatic sampler performs optimally in many applications, there are scenarios in which the Chromatic sampler can be inefficient or where we may favor more control over the execution order at the expense of added complexity. If the time required to draw from conditions differs substantially it is easy to construct a simple scenario where $p-1$ processors wait unnecessarily even when the optimal coloring is used. Alternatively, we may want to sample variables at different frequencies or use adaptive prioritization mechanisms like those proposed in Levine and Casella [2006]. In this section we construct the Asynchronous Gibbs Sampler, which is the parallel equivalent of the Random Scan Gibbs sampler.

In the Asynchronous Gibbs Sampler, each processor randomly draws the next variable to sample from some (potentially adaptive subject to the conditions in Levine and Casella [2006]) distribution. Before a processor can sample a new value for that variable, it must first acquire the corresponding **Markov Blanket Lock** using the Blanket Locking (MBL) Protocol. The Blanket Locking protocol associates a Read/Write lock with each variable in the model. The Markov blanket lock for variable X_v is obtained by acquiring the read-locks on all neighboring variables $X_{\mathcal{N}_v}$ and the write lock on X_v. To ensure that the Blanket Locking Protocol is Dead-lock free, locks are always acquired and released using the canonical ordering of the variables. The Markov blanket locking protocol (MBL) is actually equivalent to the locking protocol used in the LoopyBP `SendMessages` routine described in Alg. 3.5.

We show that the Asynchronous Gibbs sampler is ergodic and equivalent to the sequential execution of the Random Scan Gibbs sampler. Ergodicity relies on the notion of Sequential Consistency. The locking strategy employed by the Asynchronous Gibbs sampler ensures that there exists a sequential execution which recovers the same output as every parallel execution. Alternatively, since we lock the Markov Blanket, we can show that all variables sampled simultaneously are also conditionally independent given the assignment to the remaining variables.

One of the major drawback to the asynchronous Gibbs sampler is that processors can collide on locks, resulting in sequentialization and leading to a loss in parallel performance. We can, none the less, provide an encouraging lower bound on the expected number of processors that may operate simultaneously (i.e., execute without colliding on a lock). If we consider a small grid ($d=4$) Ising model with $n=20^2$ variables then using $p=8$ processors, Theorem 4.4.1 states that in expectation we will be able to use at least 7 of the processors in parallel. The intuition behind the bound in Theorem 4.4.1 is that at least one processor will always run and the remaining $p-1$ remaining processors run successfully with probability at most $(1-(p-1)(d+1)/n)$.

Theorem 4.4.1 (Randomized Asynchronous Parallelism)**.** *If p processors simultaneously try to grab variables uniformly at random from an MRF with n random variables and maximum degree d then the expected number of processors which will be able to execute in parallel is at least:*

$$\mathbf{E}\left[\textit{\# Processors Running in Parallel}\right] \quad \geq \quad 1+(p-1)\left(1-(p-1)\left(\frac{d+1}{n}\right)\right) \tag{4.3}$$

Proof. A processor is blocked if another processor is running on that vertex or any of its neighbors $(d+1)/n$ and any of the $(p-1)$ other processors can block a vertex. Therefore a lower bound on the probability that one of the $(p-1)$ processors will succeed is given by $(1-(p-1)(d+1)/n)$. □

4.5 The Splash Gibbs Sampler

The Chromatic sampler provides a linear speedup for single-chain sampling, advancing the Markov chain for a k-colorable model in time $O\left(\frac{n}{p}+k\right)$ rather than $O(n)$. Unfortunately, some models have strongly

Algorithm 4.3: Parallel Splash Sampler

```
Input: Maximum treewidth $w_{\max}$
Input: Maximum Splash size $h_{\max}$
while $t \leq \infty$ do
    // Make $p$ bounded treewidth Splashes
    $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p} \leftarrow$ ParSplash($w_{max}, h_{max}, x^{(t)}$);
    // Calibrate each junction trees
    $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p} \leftarrow$ ParCalibrate($x^{(t)}, \{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p}$);
    // Sample each Splash
    $\{x_{\mathcal{S}_i}\}_{i=1}^{p} \leftarrow$ ParSample($\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p}$);
    // Advance the chain
    $x^{(t+1)} \leftarrow \left\{x_{\mathcal{S}_1}, ..., x_{\mathcal{S}_1}, x^{(t)}_{\neg \bigcup_{i=1}^{p} \mathcal{S}_i}\right\}$
```

correlated variables and complex dependencies, which can cause the Chromatic sampler to mix prohibitively slowly.

In the single processor setting, a common method to accelerate a slowly mixing Gibbs sampler is to introduce blocking updates [Barbu and Zhu, 2005, Hamze and de Freitas, 2004, Jensen and Kong, 1996]. In a blocked Gibbs sampler, blocks of strongly coupled random variables are sampled jointly conditioned on their combined Markov blanket. The blocked Gibbs sampler improves mixing by enabling strongly coupled variables to update jointly when individual conditional updates would force the chain to move through low-probability states in-order to mix or reach high-probability states.

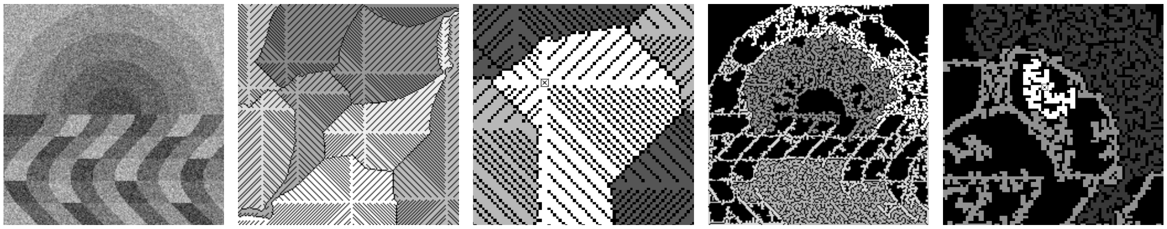
To improve mixing in the parallel setting we introduce the **Splash** sampler (Alg. 4.3), a general purpose blocking sampler. For each joint sample, the Splash sampler exploits parallelism both to construct multiple random blocks, called Splashes, and to accelerate the joint sampling of each Splash. To ensure each Splash can be safely and efficiently sampled in parallel, we developed a novel Splash generation algorithm which incrementally builds multiple conditionally independent bounded treewidth junction trees for every new sample. In the initial rounds of sampling, the Splash algorithm uses a simple adaptation heuristic which groups strongly dependent variables together based on the state of the chain. Adaptation is then disabled after a finite number of rounds to ensure ergodicity.

We present the Splash sampler in three parts. First, we present the parallel algorithm used to construct multiple conditionally independent Splashes. Next, we describe the parallel junction tree sampling procedure used to jointly sample all variables in a Splash. Finally, we present our Splash adaptation heuristic which sets the priorities used during Splash generation.

4.5.1 Parallel Splash Generation

The Splash generation algorithm (Alg. 4.4) uses p processors to incrementally build p disjoint Splashes in parallel. Each processor grows a Splash rooted at a unique vertex in the MRF (Line 1). To preserve ergodicity we require that no two roots share a common edge in the MRF, and that every variable is a root infinitely often.

Each Splash is grown incrementally using a best first search (BeFS) of the MRF. The exact order in



(a) Noisy (b) FIFO Splashes (c) FIFO Splashes (+) (d) Priority Splashes (e) Priority Splashes (+)

Figure 4.2: Different Splashes constructed on a 200 × 200 image denoising grid MRF. **(a)** A noisy sunset image. Eight Splashes of treewidth 5 were constructed using the FIFO **(b)** and priority **(d)** ordering. Each splash is shown in a different shade of gray and the black pixels are not assigned to any Splash. The priorities were obtained using the adaptive heuristic. In **(c)** and (e) we zoom in on the Splashes to illustrate their structure and the black pixels along the boundary needed to maintain conditional independence.

which variables are explored is determined by the call to `NextVertexToExplore(`$\mathcal{B}$`)` on Line 2 of Alg. 4.4 which selects (and removes) the next vertex from the boundary $\mathcal{B}$. In Figure 4.2 we plot several simultaneous Splashes constructed using a first-in first-out (FIFO) ordering (Figure 4.2b) and a prioritized ordering (Figure 4.2d).

The Splash boundary is extended until there are no remaining variables that can be safely added or the Splash is sufficiently large. A variable cannot be safely added to a Splash if sampling the resulting Splash is excessively costly (violates a treewidth bound) or if the variable or any of its neighbors are members of other Splashes (violates conditional independence of Splashes).

There are two core challenges to constructing multiple bounded computation Splashes in parallel. First, we need a fast incremental algorithm to estimate the computational cost of sampling a Splash; and second, we need a deadlock-free, race-free protocol to ensure that simultaneously constructed Splashes are conditionally independent.

To bound the computational complexity of sampling, and to jointly sample the Splash, we rely on junction trees. A junction tree, or clique graph, is an undirected acyclic graphical representation of the joint distribution over a collection of random variables. For a Splash containing the variables $X_{\mathcal{S}}$, we construct a junction tree $(\mathcal{C}, E) = \mathbf{J}_{\mathcal{S}}$ representing the conditional distribution $\mathbf{P}\left(X_{\mathcal{S}} \mid x_{-\mathcal{S}}\right)$. The set of vertices $\mathbf{C} \in \mathcal{C}$ is called a clique and is a subset of the vertices (i.e., $\mathbf{C} \subseteq \mathcal{S}$) in the Splash $\mathcal{S}$. The cliques satisfy the constraint that for every factor domain $\mathbf{A} \in \mathcal{F}$ there exists a clique $\mathbf{C} \in \mathcal{C}$ such that $\mathbf{A} \cap \mathcal{S} \subseteq \mathbf{C}$. The edges E of the junction tree satisfy the running intersection property (RIP) which ensures that all cliques sharing a common variable form a connected tree.

The computational complexity of inference, and consequently sampling using a junction tree, is exponential in the treewidth; one less than number of variables in the largest clique. Therefore, to evaluate the computational cost of adding a new variable X_v to the Splash, we need an efficient method to extend the junction tree $\mathbf{J}_{\mathcal{S}}$ over $X_{\mathcal{S}}$ to a junction tree $\mathbf{J}_{\mathcal{S}+v}$ over $X_{\mathcal{S}\cup\{v\}}$ and evaluate the resulting treewidth.

To efficiently build incremental junction trees, we developed a novel junction tree extension algorithm (Alg. 4.5) which emulates standard variable elimination, with variables being eliminated in the reverse of the order they are added to the Splash (e.g., if X_i is added to $\mathbf{J}_{\mathcal{S}}$ then X_i is eliminated before all $X_{\mathcal{S}}$). Because each Splash grows outwards from the root, the resulting elimination ordering is optimal on tree MRFs and typically performs well on cyclic MRFs.

Algorithm 4.4: ParallelSplash: Parallel Splash Generation

```
Input: Maximum treewidth $w_{\text{max}}$
Input: Maximum Splash size $h_{\text{max}}$
Output: Disjoint Splashes $\{\mathcal{S}_1, \ldots, \mathcal{S}_p\}$
do in parallel $i \in \{1, \ldots, p\}$
    $r \leftarrow$ NextRoot($i$) // Unique roots
    $\mathcal{S}_i \leftarrow \{r\}$ // Add $r$ to splash
    $\mathcal{B} \leftarrow \mathcal{N}_r$ // Add neighbors to boundary
    $\mathcal{V} \leftarrow \{r\} \cup \mathcal{N}_r$ // Visited vertices
    $\mathbf{J}_{\mathcal{S}_i} \leftarrow$ JunctionTree($\{r\}$)
    while $(|\mathcal{S}_i| < h_{max}) \bigwedge (|\mathcal{B}| > 0)$ do
        $v \leftarrow$ NextVertexToExplore($\mathcal{B}$)
        MarkovBlanketLock($X_v$)
        // Check that $v$ and its neighbors $\mathcal{N}_v$ are not in other
           Splashes.
        safe $\leftarrow \left|(\{v\} \cup \mathcal{N}_v) \cap \left(\bigcup_{j \neq i} \mathcal{S}_j\right)\right| = 0$
        $\mathbf{J}_{\mathcal{S}_i+v} \leftarrow$ ExtendJunctionTree($\mathbf{J}_{\mathcal{S}_i}$, $v$)
        if safe $\bigwedge$ TreeWidth($\mathbf{J}_{\mathcal{S}_i+v}$) $< w_{max}$ then
            $\mathbf{J}_{\mathcal{S}_i} \leftarrow \mathbf{J}_{\mathcal{S}_i+v}$ // Accept new tree
            $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup \{v\}$
            $\mathcal{B} \leftarrow \mathcal{B} \cup (\mathcal{N}_v \backslash \mathcal{V})$ // Extend boundary
            $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{N}_v$ // Mark visited
        MarkovBlanketFree($X_v$)
```

The incremental junction tree extension algorithm (Alg. 4.5) begins by eliminating X_i and forming the new clique $\mathbf{C}_i = (\mathcal{N}_i \cap \mathcal{S}) \cup \{i\}$ which is added to $\mathbf{J}_{\mathcal{S}+i}$. We then attach $\mathbf{C}_i$ to the *most recently added* clique $\mathbf{C}_{\texttt{Pa}[i]}$ that contains a variable in $\mathbf{C}_i$ ($\mathbf{C}_{\texttt{Pa}[i]}$ denotes the parent of $\mathbf{C}_i$). The running intersection property (RIP) is then restored by propagating the newly added variables back up the tree. We define the remaining unsatisfied variables $\mathcal{R} = \mathbf{C}_i \backslash \{i\}$ and insert $\mathcal{R}$ into its parent clique $\mathbf{C}_{\texttt{Pa}[i]}$. The RIP condition is now satisfied for variables in $\mathcal{R}$ which were already in $\mathbf{C}_{\texttt{Pa}[i]}$. The parent for $\mathbf{C}_{\texttt{Pa}[i]}$ is then recomputed, and any unsatisfied variables are propagated up the tree in the same way. We demonstrate this algorithm with a simple example in Figure 4.3.

To ensure that simultaneously constructed Splashes are conditionally independent, we employ the Markov blanket locking (MBL) protocol from the Asynchronous Gibbs sampler. Once the `MarkovBlanketLock(`X_v`)` has been acquired, no other processor can assign X_v or any of it neighbors $X_{\mathcal{N}_v}$ to a Splash. Therefore, we can safely test if X_v or any of its neighbors $X_{\mathcal{N}_v}$ are currently assigned to other Splashes. Since we only add X_v to the Splash if both X_v and all its neighbors are currently unassigned to other Splashes, there will never be an edge in the MRF that connects two Splashes. Consequently, simultaneously constructed Splashes are conditionally independent given all remaining unassigned variables.

Algorithm 4.5: ExtendJunctionTree Algorithm

```
Input: The original junction tree $(\mathcal{C}, E) = \mathbf{J}_{\mathcal{S}}$.
Input: The variable $X_i$ to add to $\mathbf{J}_{\mathcal{S}}$
Output: $\mathbf{J}_{\mathcal{S}+i}$
Define: $\mathbf{C}_u$ as the clique created by eliminating $u \in \mathcal{S}$
Define: $\texttt{V}[\mathbf{C}] \in \mathcal{S}$ as the variable eliminated when creating $\mathbf{C}$
Define: $\texttt{t}[v]$ as the time $v \in \mathcal{S}$ was added to $\mathcal{S}$
Define: $\texttt{Pa}[v] \in \mathcal{N}_v \cap \mathcal{S}$ as the next neighbor of $v$ to be eliminated.
$\mathbf{C}_i \leftarrow (\mathcal{N}_i \cap \mathcal{S}) \cup \{i\}$
$\texttt{Pa}[i] \leftarrow \arg\max_{v \in \mathbf{C}_i \backslash \{i\}} \texttt{t}[v]$
// ----------- Repair RIP -------------
$\mathcal{R} \leftarrow \mathbf{C}_i \backslash \{i\}$ // RIP Set
$v \leftarrow \texttt{Pa}[i]$
while $|\mathcal{R}| > 0$ do
    $\mathbf{C}_v \leftarrow \mathbf{C}_v \cup \mathcal{R}$ // Add variables to parent
    $w \leftarrow \arg\max_{w \in \mathbf{C}_v \backslash \{v\}} \texttt{t}[w]$ // Find new parent
    if $w = \texttt{Pa}[v]$ then
        $\mathcal{R} \leftarrow (\mathcal{R} \backslash \mathbf{C}_i) \backslash \{i\}$
    else
        $\mathcal{R} \leftarrow (\mathcal{R} \cup \mathbf{C}_i) \backslash \{i\}$
        $\texttt{Pa}[v] \leftarrow w$ // New parent
    $v \leftarrow \texttt{Pa}[v]$ // Move upwards
```

4.5.2 Parallel Splash Sampling

Once we have constructed p conditionally independent Splashes $\{\mathcal{S}_1\}_{i=1}^{p}$, we jointly sample each Splash by drawing from $\mathbf{P}\left(X_{\mathcal{S}_i} \,|\, x_{-\mathcal{S}_i}\right)$ in parallel. Sampling is accomplished by first calibrating the junction trees $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p}$, and then running backward-sampling starting at the root to jointly sample all the variables in each Splash. We also use the calibrated junction trees to construct Rao-Blackwellized marginal estimators. If the treewidth or the size of each Splash is large, it may be beneficial to construct fewer Splashes, and instead assign multiple processors to accelerate the calibration and sampling of individual junction trees.

To calibrate the junction tree we use the `ParCalibrate` function (Alg. 4.6). The `ParCalibrate` function constructs all clique potentials in parallel by computing the products of the assigned factors conditioned on the variables not in the junction tree. Finally, parallel belief propagation is used to calibrate the tree by propagating messages in parallel following the optimal forward-backward schedule. At each level individual message computations are done in parallel.

Parallel backward-sampling is accomplished by the function `ParSample` which takes the calibrated junction tree and draws a new joint assignment in parallel. The `ParSample` function begins by drawing a new joint assignment for the root clique using the calibrated marginal. Then, in parallel, each child is sampled conditioned on the parent assignment and the messages from the children.

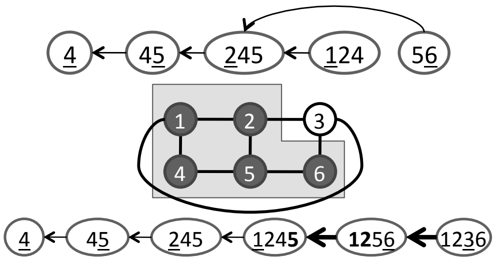


Figure 4.3: **Incremental Junction Tree Example**: The junction tree on the top comprises the subset of variables {1,2,4,5,6} of the MRF (center). The tree is formed by the variable elimination ordering {6,1,2,5,4} (reading the underlined variables of the tree in reverse). To perform an incremental insertion of variable 3, we first create the clique formed by the elimination of 3 ({1,2,3,6}) and insert it into the end of the tree. Its parent is set to the latest occurrence of any of the variables in the new clique. Next the set {1,2,6} is inserted into its parent (boldface variables), and its parent is recomputed in the same way.

Algorithm 4.6: ParallelCalibrate: Junction Calibration

```
Input: $x^{(t)}$, $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p}$
do in parallel processor $i \in \{1, \ldots, p\}$
    forall Cliques $c \in \mathbf{J}_{\mathcal{S}_i}$ do in parallel
        // Initialize clique potentials
        $\psi_c(X_c) \leftarrow \prod_{f_\alpha \in c} f_\alpha(X_c, x_{-c}^{(t)})$
    $\mathbf{J}_{\mathcal{S}_i} \leftarrow$ ParallelBPCalibrate($\{\psi_c\}_{c \in \mathcal{S}_i}$)
Output: Calibrated junction trees $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p}$
```

4.5.3 Adaptive Splash Generation

As discussed earlier, the order in which variables are explored when constructing a Splash is determined on Line 2 in the `ParSplash` algorithm (Alg. 4.4). We propose a simple adaptive prioritization heuristic, based on the current assignment to $x^{(t)}$, that prioritizes variables at the boundary of the current tree which are strongly coupled with variables already in the Splash. We assign each variable $X_v \in \mathcal{B}$ a score using the likelihood ratio:

$$\mathbf{s}[X_v] = \left\| \log \frac{\sum_x \mathbf{P}\left(X_{\mathcal{S}}, X_v = x \,|\, X_{-\mathcal{S}} = x_{-\mathcal{S}}^{(t)}\right)}{\mathbf{P}\left(X_{\mathcal{S}}, X_v = x_v^{(t)} \,|\, X_{-\mathcal{S}} = x_{-\mathcal{S}}^{(t)}\right)} \right\|_1, \tag{4.4}$$

and includes the variable with the highest score. Effectively, Eq. (4.4) favors variables with greater average log likelihood than conditional log likelihood. We illustrate the consequence of applying this metric to an image denoising task in which we denoise the synthetic image shown in Figure 4.2a. In Figure 4.4 we show a collection of Splashes constructed using the score function Eq. (4.4). To see how priorities evolve over time, we plot the update frequencies early (Figure 4.4a) and later (Figure 4.4b) in the execution of the Splash scheduler.

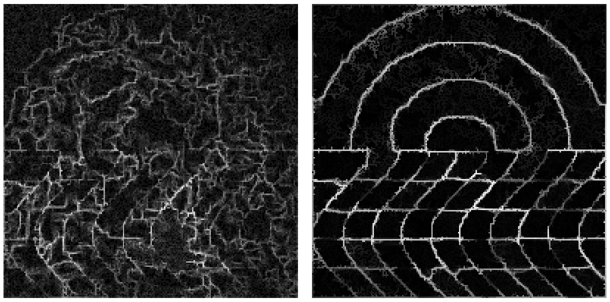

(a) Early (b) Later

Figure 4.4: The update frequencies of each variable in the 200×200 image denoising grid MRF for the synthetic noisy image shown in Figure 4.2a. The brighter pixels have been prioritized higher and are therefore updated more frequently. **(a)** The early update counts are relatively uniform as the adaptive heuristic has not converged on the priorities. **(b)** The final update counts are focused on the boundaries of the regions in the model corresponding to pixels that can be most readily changed by blocked steps.

4.5.4 A Need for Vanishing Adaptation

Given the flexibility in scheduling Geman and Geman [1984] ascribed to the Gibbs sampler, it would seem reasonable to continually adapt the update schedule and Splash shape especially since any individual Splash preserves the underlying distribution. Indeed, Levine and Casella [2006] make a similar claim about their Algorithm 7.1: which is a generalization of adaptive Gibbs samplers. Unfortunately, we can demonstrate, by means of a simple counter example, that tuning the Splash (blocking) in general breaks ergodicity even when we include all variables with positive probability.

Consider the case of two uniform independent binary variables X_1 and X_2 with constant probability mass function $\mathbf{P}(x_1, x_2) \propto 1$. We define an adaptive sampling procedure which picks a variable to sample uniformly at random if both variables have the same assignment ($x_1 = x_2$). However, if the variables have different assignments ($x_1 \neq x_2$), we pick the variable with assignment 1 with 90% probability. Let the choice of variable to sample (Splash) be represented by $\mathbf{P}(\mathcal{S} \mid X_1, X_2)$ where $\mathcal{S} \subseteq \{1, 2\}$. Let (X_1, X_2) be the current state and (X_1', X_2') be the new state after one step of the procedure. Then the transition kernel is given by (where $\mathbf{1}[\cdot]$ is the indicator function):

$$\begin{aligned}
\mathbf{P}\left(X_1', X_2' \,|\, \mathcal{S} = \{1\}, X_1, X_2\right) &\propto \mathbf{1}\left[x_2' = x_2\right] \\
\mathbf{P}\left(X_1', X_2' \,|\, \mathcal{S} = \{2\}, X_1, X_2\right) &\propto \mathbf{1}\left[x_1' = x_1\right]
\end{aligned}$$

Marginalizing the assignments to (X_1, X_2) with respect to the true probability $\mathbf{P}(X_1, X_2)$ we obtain:

$$\begin{aligned}
\mathbf{P}\left(X_1' = x_1', X_2' = x_2'\right) &= \sum_{x_1,x_2}\sum_{\mathcal{S}} \mathbf{P}\left(\mathcal{S} \,|\, X_1, X_2\right)\mathbf{P}\left(X_1', X_2' \,|\, \mathcal{S}, X_1, X_2\right)\mathbf{P}\left(X_1, X_2\right) \\
&\propto \sum_{x_1,x_2}\sum_{\mathcal{S}} \mathbf{P}\left(\mathcal{S} \,|\, X_1, X_2\right)\mathbf{P}\left(X_1', X_2' \,|\, \mathcal{S}, X_1, X_2\right) \\
&\propto \sum_{x_1,x_2} \mathbf{P}\left(\mathcal{S} = \{1\} \,|\, X_1, X_2\right)\mathbf{1}\left[x_2' = x_2\right] + \mathbf{P}\left(\mathcal{S} = \{2\} \,|\, X_1, X_2\right)\mathbf{1}\left[x_1' = x_1\right] \\
&\propto \left(\sum_{x_1} \mathbf{P}\left(\mathcal{S} = \{1\} \,|\, X_1, X_2 = x_2'\right)\right) + \left(\sum_{x_2} \mathbf{P}\left(\mathcal{S} = \{2\} \,|\, X_1 = x_1', X_2\right)\right) \\
&= \mathbf{1}\left[x_2' = 1\right](0.5 + 0.1) + \mathbf{1}\left[x_2' = 0\right](0.9 + 0.1) \\
&\qquad + \mathbf{1}\left[x_1' = 1\right](0.5 + 0.1) + \mathbf{1}\left[x_1' = 0\right](0.9 + 0.1),
\end{aligned}$$

where the last step of the process is computing by substituting in the values of $\mathbf{P}\left(\mathcal{S} \,|\, X_1, X_2\right)$ as defined by the adaptation process.

The resulting distribution of $\mathbf{P}\left(X_1', X_2'\right)$ is

x_1'	x_2'	$\mathbf{P}\left(X_1' = x_1', X_2' = x_2'\right)$
0	0	0.35
0	1	0.25
1	0	0.25
1	1	0.15

which clearly is not the correct stationary distribution and the Gibbs sampler is therefore not ergodic. This counter example refutes the proof of Algorithm 7.1 in Levine and Casella [2006], and places conditions upon the Gibbs sampler described in [Geman and Geman, 1984]. Specifically, simply sampling from every variable infinitely often is insufficient for ergodicity.

To ensure the chain remains ergodic, we disable prioritized tree growth after a finite number of iterations, and replace it with a random tree growth. This form of **vanishing adaptation** helps move to high-likelihood samples and can accelerate early mixing (burn-in). However, ultimately we would like to develop adaptation heuristics that do not require vanishing adaptation and believe this is an important area of future research.

By employing vanishing adaptation we show that our Splash sampler with vanishing is ergodic:

Theorem 4.5.1 (Splash Sampler Ergodicity). *The adaptive Splash sampler with vanishing adaptation is ergodic and converges to the true distribution.*

Proof. We prove Theorem 4.5.1 in three parts. First we show that $\mathbf{P}(X)$ is the invariant distribution of the Splash Sampler:

$$\mathbf{P}\left(X^{(t+1)}\right) = \sum_{x} K\left(X^{(t+1)} \,|\, X^{(t)} = x\right)\mathbf{P}\left(X^{(t)} = x\right) \tag{4.5}$$

Second, we show that the Splash sampler forgets its starting state:

$$\sup_{x,y,z}\left|\left|\mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = y\right) - \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = z\right)\right|\right|_1 \leq \gamma^{t/n} \tag{4.6}$$

Finally, we show that the sampler draws from $\mathbf{P}(X)$ in the limit:

$$\lim_{t\to\infty} \sup_{y,x} \left|\left|\mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = y\right) - \mathbf{P}(X = x)\right|\right|_1 = 0 \tag{4.7}$$

Distributional Invariance: Because vanishing adaptation is used we show that $\mathbf{P}(X)$ is preserved for Splash updates without adaptation: where the choice of Splash $\mathcal{S}^{(t)}$ does not depend on the state of the chain. However, we do allow the choice of the Splash to depend on all previous Splash choices $\left\{\mathcal{S}^{(0)}, \ldots, \mathcal{S}^{(t)}\right\}$. The transition kernel for the Splash sampler given the Splash $\mathcal{S}$ is defined as:

$$K\left(X^{(t+1)} \,|\, x^{(t)}\right) = \sum_{\mathcal{S}} \mathbf{P}\left(\mathcal{S} \,|\, \mathcal{S}^{(0)} \ldots \mathcal{S}^{(t-1)}\right) \mathbf{1}\left[X_{-\mathcal{S}}^{(t+1)} = X_{-\mathcal{S}}^{(t)}\right] \mathbf{P}\left(X_{\mathcal{S}}^{(t+1)} \,|\, X_{-\mathcal{S}}^{(t+1)}\right) \tag{4.8}$$

Substituting Eq. (4.8) into Eq. (4.5) we obtain:

$$\sum_{X^{(t)}} K\left(X^{(t+1)} \,|\, X^{(t)}\right) \mathbf{P}\left(X^{(t)}\right) \tag{4.9}$$

$$= \sum_{X^{(t)}} \sum_{\mathcal{S}} \mathbf{P}\left(\mathcal{S} \,|\, \mathcal{S}^{(0)} \ldots \mathcal{S}^{(t)}\right) \mathbf{1}\left[x_{-\mathcal{S}}^{(t+1)} = x_{-\mathcal{S}}^{(t)}\right] \mathbf{P}\left(X_{\mathcal{S}}^{(t+1)} \,|\, X_{-\mathcal{S}}^{(t+1)}\right) \mathbf{P}\left(X^{(t)}\right) \tag{4.10}$$

$$= \sum_{\mathcal{S}} \mathbf{P}\left(\mathcal{S} \,|\, \mathcal{S}^{(0)} \ldots \mathcal{S}^{(t)}\right) \mathbf{P}\left(X_{\mathcal{S}}^{(t+1)} \,|\, X_{-\mathcal{S}}^{(t+1)}\right) \sum_{X^{(t)}} \mathbf{1}\left[x_{-\mathcal{S}}^{(t+1)} = x_{-\mathcal{S}}^{(t)}\right] \mathbf{P}\left(X^{(t)}\right) \tag{4.11}$$

$$= \sum_{\mathcal{S}} \mathbf{P}\left(\mathcal{S} \,|\, \mathcal{S}^{(0)} \ldots \mathcal{S}^{(t)}\right) \mathbf{P}\left(X_{\mathcal{S}}^{(t+1)} \,|\, X_{-\mathcal{S}}^{(t+1)}\right) \mathbf{P}\left(X_{-\mathcal{S}}^{(t+1)}\right) \tag{4.12}$$

$$= \sum_{\mathcal{S}} \mathbf{P}\left(\mathcal{S} \,|\, \mathcal{S}^{(0)} \ldots \mathcal{S}^{(t)}\right) \mathbf{P}\left(X^{(t+1)}\right) \tag{4.13}$$

$$= \mathbf{P}\left(X^{(t+1)}\right) \tag{4.14}$$

Dependence on Starting State: We now show that the Splash sampler forgets its initial starting state. This is done by first showing that there is a positive probability of reaching any state after a bounded number of Splash operations. Then we use this strong notion of irreducibility to setup a recurrence which bounds the dependence on the starting state.

The Splash algorithm sweeps across the roots ensuring that after $\Delta t \leq n$ tree updates all variables are sampled at least once. Define the time t_i as the last time variable X_i was updated. Without loss of generality (by renumbering the variables) we assume that $t_1 < t_2 < \ldots < t_n$. We can then bound the probability that after n Splashes we reach state x given we were initially at state y,

$$\mathbf{P}\left(X^{(n)} = x \,|\, X^{(0)} = y\right) \geq \prod_{i=1}^{n} \inf_{x_{-i}} \mathbf{P}\left(X_i = x_i \,|\, X_{-i} = x_{-i}\right). \tag{4.15}$$

Define the smallest conditional probability,

$$\delta = \inf_{i,x} \mathbf{P}\left(X_i = x_i \,|\, X_{-i} = x_{-i}\right), \tag{4.16}$$

then we can lower bound the probability of reach any state:

$$\mathbf{P}\left(X^{(n)} = x \,|\, X^{(0)} = y\right) \geq \delta^n. \tag{4.17}$$

Effectively, we are stating that $\mathbf{P}\left(X\right)$ is irreducible since all the conditionals are positive. As a consequence we can show that there exists an γ such that $0 \leq \gamma < 1$ and

$$\sup_{x,y,z} \left|\left|\mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = y\right) - \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = z\right)\right|\right|_1 \leq \gamma^{t/n}. \tag{4.18}$$

This is trivially true for $t = 0$. We can rewrite the left side of the above equation as

$$\sup_{x,y,z} \left|\left|\mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = y\right) - \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = z\right)\right|\right|_1 =$$

$$\sup_{x} \left|\left|\sup_{y} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = y\right) - \inf_{z} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = z\right)\right|\right|_1. \tag{4.19}$$

Now we will bound the inner sup and inf terms. For $t > n$ we can introduce a probability measure μ such that

$$\begin{aligned}
&\sup_{y} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = y\right) \\
&\quad = \sup_{y} \sum_{w} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w\right) \mathbf{P}\left(X^{(n)} = w \,|\, X^{(0)} = y\right) && (4.20) \\
&\quad \leq \sup_{\mu \geq \delta^n} \sum_{w} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w\right) \mu(w) && (4.21)
\end{aligned}$$

Then by defining the w^* as

$$w^* = \arg\sup_{w} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w\right), \tag{4.22}$$

we can easily construct the maximizing μ which places the minimal mass δ^n on all w except w^* and places the remaining mass $1 - (|\Omega| - 1)\delta^n$ on w^*. This leads to

$$\begin{aligned}
\sup_{y} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = y\right) \quad \leq \quad & (1 - (|\Omega| - 1)\delta^n)\, \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w^*\right) + && (4.23) \\
& \delta^n \sum_{w \neq w^*} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w\right). && (4.24)
\end{aligned}$$

Similarly we define w_* as the minimizing element in Ω

$$w_* = \arg\inf_{w} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w\right). \tag{4.25}$$

We can then construct the lower bound,

$$\begin{aligned}
\inf_{z} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(0)} = z\right) \quad \geq \quad & (1 - (|\Omega| - 1)\delta^n)\, \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w_*\right) + && (4.26) \\
& \delta^n \sum_{w \neq w^*} \mathbf{P}\left(X^{(t)} = x \,|\, X^{(n)} = w\right). && (4.27)
\end{aligned}$$

Taking the difference, we get

$$\begin{aligned}
&\sup_{x,y,z}\left|\left|\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=y\right)-\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=z\right)\right|\right|_1\\
&\quad=\sup_x\left|\left|\sup_y\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=y\right)-\inf_z\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=z\right)\right|\right|_1 &(4.28)\\
&\quad\le\sup_x||\left(1-(|\Omega|-1)\delta^n\right)\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=w^*\right)+\delta^n\sum_{w\neq w^*}\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=w\right)-\\
&\qquad\qquad\left(1-(|\Omega|-1)\delta^n\right)\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=w_*\right)-\delta^n\sum_{w\neq w_*}\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=w\right)||_1. &(4.29)
\end{aligned}$$

We can bound the sum terms

$$\begin{aligned}
&\delta^n\sum_{w\neq w^*}\mathbf{P}\left(X(t)=x\,|\,X(n)=w\right)-\delta^n\sum_{w\neq w_*}\mathbf{P}\left(X(t)=x\,|\,X(n)=w\right)\le\\
&\qquad\delta^n\sum_{w}\mathbf{P}\left(X(t)=x\,|\,X(n)=w\right)-\delta^n\sum_{w}\mathbf{P}\left(X(t)=x\,|\,X(n)=w\right)=0,
\end{aligned}$$

and then simplify Eq. (4.29) to obtain

$$\begin{aligned}
&\sup_{x,y,z}\left|\left|\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=y\right)-\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=z\right)\right|\right|_1\le\\
&\qquad\left(1-(|\Omega|-1)\delta^n\right)\sup_{x,y,z}\left|\left|\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=y\right)-\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=z\right)\right|\right|_1. \qquad (4.30)
\end{aligned}$$

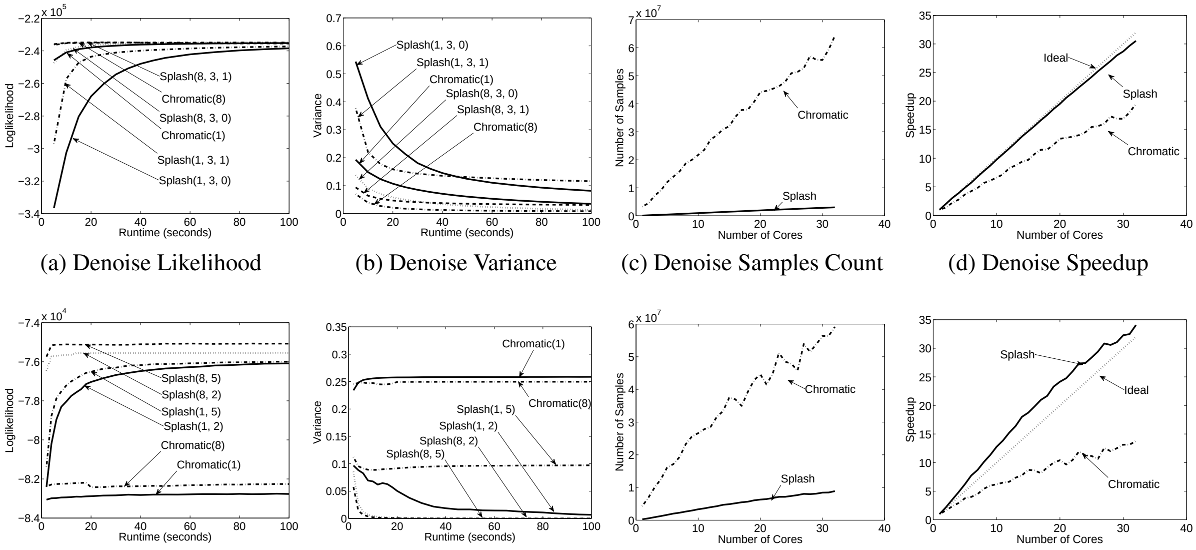
We can repeat this procedure t/n times to the term $\left|\left|\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=y\right)-\mathbf{P}\left(X^{(t)}=x\,|\,X^{(n)}=z\right)\right|\right|_1$ on the right side of the above equation to obtain the desired result,

$$\sup_{x,y,z}\left|\left|\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=y\right)-\mathbf{P}\left(X^{(t)}=x\,|\,X^{(0)}=z\right)\right|\right|_1\le\left(1-(|\Omega|-1)\delta^n\right)^{t/n}. \qquad (4.31)$$

We can use Eq. (4.31) along with π invariance to finish the proof,

$$\begin{aligned}
&\lim_{t\to\infty}\sup_{y,x}\left|\left|\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=x\right)-\mathbf{P}\left(y\right)\right|\right|_1\\
&\quad=\lim_{t\to\infty}\sup_{y,x}\left|\left|\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=x\right)-\sum_z\mathbf{P}\left(y\,|\,z\right)\mathbf{P}\left(z\right)\right|\right|_1 &(4.32)\\
&\quad=\lim_{t\to\infty}\sup_{y,x}\left|\left|\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=x\right)-\sum_z\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=z\right)\mathbf{P}\left(z\right)\right|\right|_1 &(4.33)\\
&\quad=\lim_{t\to\infty}\sup_{y,x}\left|\left|\sum_z\mathbf{P}\left(z\right)\left(\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=x\right)-\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=z\right)\right)\right|\right|_1 &(4.34)\\
&\quad\le\lim_{t\to\infty}\sup_{y,x,z}\left|\left|\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=x\right)-\mathbf{P}\left(X^{(t)}=y\,|\,X^{(0)}=z\right)\right|\right|_1 &(4.35)\\
&\quad\le\lim_{t\to\infty}\left(1-(|\Omega|-1)\delta^n\right)^{t/n} &(4.36)\\
&\quad=0. &(4.37)
\end{aligned}$$

□



(a) Denoise Likelihood (b) Denoise Variance (c) Denoise Samples Count (d) Denoise Speedup

(e) CORA-1 Likelihood (f) CORA-1 Variance (g) CORA-1 Samples Count (h) CORA-1 Speedup

Figure 4.5: Comparison of Chromatic sampler and the Splash sampler at different settings (i.e., Chromatic(p), Splash($p, w_{\max}$, adaptation) for p processors and treewidth $w_{\max}$) on the synthetic image denoising grid model and the Cora Markov logic network. Adaptation was not used in the CORA-1 MLN. **(a,e)** The un-normalized log-likelihood plotted as a function of running-time. **(b,f)** The variance in the estimator of the expected assignment computed across 10 independent chains with random starting points. **(c,g)** The total number of variables sampled in a 20 second window plotted as a function of the number of cores. **(d,h)** The speedup in number of samples drawn as a function of the number of processors.

4.6 Experiments

We implemented an optimized C++ version of both the Chromatic and Splash samplers for arbitrary discrete factorized models. Although Alg. 4.3 is presented as a sequence of synchronous parallel steps, our implementation splits these steps over separate processors to maximize performance and eliminate the need for threads to join between phases. While the calibration and sampling of individual junction trees can also be parallelized, we found that for the typically small treewidth used in our experiments, the overhead associated the additional parallelism overrode any gains. Nonetheless, when we made the treewidth sufficiently large (e.g., 10^6 sized factors) we were able to obtain $13\times$-speedup in junction tree calibration on 32 cores.

To evaluate the proposed algorithms in both the weakly and strongly correlated setting we selected two representative large-scale models. In the weakly correlated setting we used a $40,000$ variable 200×200 grid MRF similar to those used in image processing. The latent pixel values were discretized into 5 states. Gibbs sampling was used to compute the expected pixel assignments for the synthetic noisy image shown in Figure 4.2a. We used Gaussian node potentials centered around the pixel observations with $\sigma^2 = 1$ and Ising-Potts edge potentials of the form $\exp(-3\delta(x_i \neq x_j))$. To test the algorithms in the strongly correlated setting, we used the CORA-1 Markov Logic Network (MLN) obtained from Domingos [2009]. This large real-world factorized model consists of over $10,000$ variables and $28,000$ factors, and has a much higher connectivity and higher order factors than the pairwise MRF.

In Figure 4.5 we present the results of running both algorithms on both models using a state-of-the-art 32 core Intel Nehalem (X7560) server with hyper-threading disabled. We plot un-normalized log-likelihood and across chain variance in-terms of wall-clock time. In Figure 4.5a and Figure 4.5e we plot the un-normalized log-likelihood of the last sample as a function of time. While in both cases the Splash algorithm out-performs the chromatic sampler, the difference is more visible in the CORA-1 MLN. We found that the adaptation heuristic had little effect in likelihood maximization on the CORA-1 MLN, but did improve performance on the denoising model by focusing the Splashes on the higher variance regions. In Figure 4.5b and Figure 4.5f we plot the variance in the expected variable assignments across 10 independent chains with random starting points. Here we observe that for the faster mixing denoising model, the increased sampling rate of the Chromatic sampler leads to a greater reduction in variance while in the slowly mixing CORA-1 MLN only the Splash sampler is able to reduce the variance.

To illustrate the parallel scaling we plot the number of samples generated in a 20 seconds (Figure 4.5c and Figure 4.5d) as well as the speedup in sample generation (Figure 4.5c and Figure 4.5d). The speedup is computed by measuring the multiple of the number of samples generated in 20 seconds using a single processor. The ideal speedup is linear with 32× speedup on 32 cores.

We find that the Chromatic sampler typically generates an order of magnitude more samples per second than the more costly Splash sampler. However, if we examine speedup curves we see that the larger cost associated with the Splash construction and inference contributes to more exploitable coarse grain parallelism. Interestingly, in Figure 4.5h we see that the Splash sampler exceeds the ideal scaling. This is actually a consequence of the high connectivity forcing each of the parallel Splashes to be smaller as the number of processors increases. As a consequence the cost of computing each Splash is reduced and the sampling rate increases. However, this also reduces some of the benefit from the Splash procedure as the size of each Splash is smaller resulting a potential increase in mixing time.

4.7 Additional Related Work

As discussed in the motivation, the majority of existing large-scale sampling efforts have not preserved ergodicity and in many cases focused on largely synchronous updates. However, some more recent sampling efforts have taken a strongly asynchronous approach. For example, Smola and Narayanamurthy [2010] constructed one of the largest collapsed Gibbs samplers for LDA. This approach is entirely asynchronous with each variable being sampled as fast as possible. While this approach has not yet been shown to converge to the correct distribution, it seems to perform extremely well in practice.

Later work by Ahmed et al. [2012] improves upon the work by Smola and Narayanamurthy [2010], and introduces the **parameter server** a powerful new primitive for expressing *highly asynchronous* but *unstructured* computation. The parameter server is a large distributed key-value store which supports *eventually consistent* [Vogels, 2009] transactions using a novel *delta update* strategy. Each processor then sequentially updates its local variables writing their new values to the parameter server. As a consequence the updates are *locally consistent* but globally inconsistent. The system them replicates these changes to the remaining processors as quickly as possible. While this work does not leverage the graphical dependency structure, the novel use of asynchrony is consistent with the GrAD methodology.

In the single processor setting there has been a lot of work on blocking Gibbs samplers [Barbu and Zhu, 2005, Hamze and de Freitas, 2004, Jensen and Kong, 1996]. Our Splash algorithm is inspired by the work of Hamze and de Freitas [2004] which studies the case of constructing a fixed set of tree-width one spanning trees. They provide deeper analysis of the potential acceleration in mixing achieved by tree based blocking with Rao-Blackwellization.

4.8 Conclusion

In this chapter we used the GrAD design methodology to develop three ergodic parallel Gibbs samplers for high-dimensional models: the Chromatic sampler, the Asynchronous Sampler, and the Splash sampler. The Chromatic parallel Gibbs sampler can be applied where single variable updates still mix well, and uses graph coloring techniques to schedule conditionally independent updates in parallel. We related Chromatic sampler to the commonly used (but non-ergodic) Synchronous Gibbs sampler, and showed that we can recover two ergodic chains from a single non-ergodic Synchronous Gibbs chain.

We relaxed the strict chromatic schedule to introduce the Asynchronous Gibbs sampler. The Asynchronous Gibbs sampler relies on the Markov Blanket Locking protocol to ensure ergodicity and actually a special case of the Splash Gibbs sampler with Splash size set to one.

In settings with tightly couples variables, the parallelism afforded by the Chromatic Gibbs sampler may be insufficient to achieve rapid mixing. We therefore proposed the Splash Gibbs sampler which incrementally constructs multiple conditionally independent bounded treewidth blocks (Splashes) in parallel. We developed a novel incremental junction tree construction algorithm which quickly and efficiently updates the junction tree as new variables are added. We further proposed a procedure to accelerate burn-in by explicitly grouping strongly dependent variables,. An interesting topic for future work is whether one can design an adaptive parallel sampler (i.e., one that *continually* modifies its behavior depending on the current state of the chain) that still retains ergodicity.

Chapter 5

GraphLab: Generalizing the GrAD Methodology

In Chapter 3 and Chapter 4 we applied the GrAD methodology to design and implement efficient parallel and distributed algorithms for probabilistic reasoning. In this chapter we generalize the thesis approach by introducing GraphLab, a high-level graph-parallel framework for the design and implementation algorithms that adhere to the GrAD methodology.

5.1 Introduction

With the exponential growth in the scale and complexity of structured probabilistic reasoning in Machine Learning and Data Mining (MLDM) applications there is an increasing need for *systems* that can execute MLDM algorithms efficiently on multicore machines and distributed environments. Simultaneously, the availability of cloud computing services like Amazon EC2 provide the promise of on-demand access to affordable large-scale computing and storage resources without substantial upfront investments. In the previous chapters we presented the design and implementation two MLDM systems for probabilistic reasoning. In both cases we had to explicitly address the design of low-level primitives for scheduling, mutual exclusion, data-placement, and distributed coordination in addition to the design of efficient algorithms.

Unfortunately, designing, implementing, and debugging the *efficient* parallel and distributed MLDM algorithms needed to fully utilize multicore and distributed systems can be extremely challenging requiring MLDM experts to address race conditions, deadlocks, distributed state, and communication protocols while simultaneously developing mathematically complex models and algorithms. While low level abstractions like MPI and pthreads provide powerful, expressive primitives, they force the user to address hardware issues and the challenges of parallel and distributed system design.

To avoid the challenges of system design, many have turned to high-level data-parallel abstractions, which dramatically simplify the design and implementation of a *restricted* class of parallel algorithms. For example, the MapReduce abstraction [Dean and Ghemawat, 2004] has been successfully applied to a broad range of ML applications [Chu et al., 2006, Panda et al., 2009, Wolfe et al., 2008, Ye et al., 2009]. The success of these abstractions derives from their ability to *restrict* the types of algorithms that may be expressed enabling the isolation of the algorithm from the systems specific challenges and forcing the algorithm designer to explicitly expose a high degree of parallelism. However, by restricting our focus to ML algorithms that are naturally expressed in existing data-parallel abstractions, we are often forced to make overly simplifying assumptions. Alternatively, as was demonstrated in Chapter 3 by coercing efficient sequential ML algorithms to satisfy the restrictions imposed by many existing abstraction, we often produce inefficient parallel algorithms that require many processors to be competitive with comparable sequential methods.

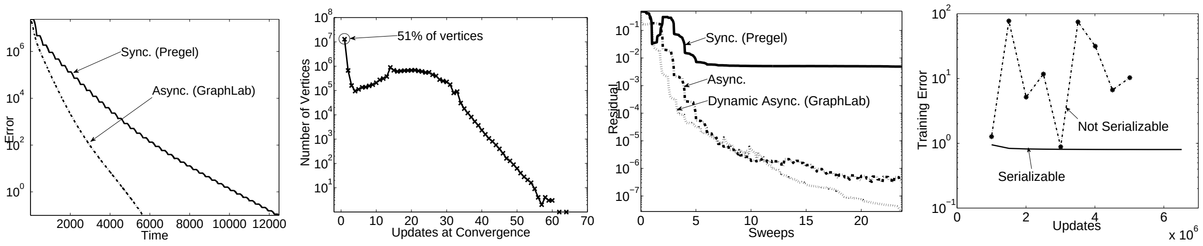
As we will demonstrate in this chapter, the efficient MLDM algorithms that follow the GrAD methodology are not naturally expressed by existing high-level data-parallel abstractions. As a consequence many [Asuncion et al., 2008, Graf et al., 2004, Nallapati et al., 2007, Newman et al., 2007, Smola and Narayanamurthy, 2010] recent large-scale learning systems are built upon low-level tools and targeted at individual models and applications. This time consuming and often redundant effort slows the progress of the field as different research groups repeatedly solve the same parallel and distributed computing problems. Therefore, the MLDM community needs a high-level distributed abstraction that specifically targets the *asynchronous*, *dynamic*, *graph-parallel* computation found in many MLDM applications while hiding the complexities of parallel and distributed system design.

In this chapter we introduce GraphLab, a new graph-parallel abstraction that explicitly targets MLDM algorithms which follow the GrAD methodology. GraphLab enables MLDM experts to easily design and implement efficient scalable parallel and distributed MLDM algorithms by composing problem specific computation, data-dependencies, and scheduling.

We begin by first providing some background on existing abstractions and their limitations in the context of the GrAD methodology. We then present the Graph-Parallel setting and introduce the GraphLab abstraction. We describe how the abstraction can be used to efficiently express a range of MLDM algorithms including the earlier Belief Propagation and Gibbs sampling algorithms. We then describe an efficient *multicore* implementation of the GraphLab abstraction and evaluate our implementation by using it to run several GrAD algorithms including belief propagation and Gibbs sampling as well as several new algorithms designed using the GraphLab abstraction and the GrAD methodology. We then extend our multicore implementation to the *distributed* (cloud) setting and discuss several methods used to achieve serializability and fault-tolerance. Finally we evaluate the distributed implementation on an EC2 cluster demonstrating order of magnitude gains of published results.

The main contributions of this chapter are:

- A summary of common properties of MLDM algorithms (Section 5.2) and the limitations of existing abstractions frameworks (Section 5.3).
- The **GraphLab** abstraction (Section 5.4) which targets algorithms designed using the GrAD methodology and is composed of:
    1. A **data graph** (Section 5.4.1) which simultaneously represents data and computational dependencies.
    2. A **vertex program** (Section 5.4.2) representation of user defined computation which isolates the algorithm from the movement of data and enables dynamic scheduling.
    3. A simple serial **execution model** (Section 5.4.3) which exposes a high degree of asynchronous parallelism.



(a) Async vs Sync PageRank (b) Dynamic PageRank (c) LoopyBP Conv. (d) ALS Serializability

Figure 5.1: **(a)** Rate of convergence, measured in L_1 error to the true PageRank vector versus time, of the PageRank algorithm on a 25M vertex 355M edge web graph on 16 processors. **(b)** The distribution of update counts after running dynamic PageRank to convergence. Notice that the majority of the vertices converged in only a single update. **(c)** Rate of convergence of Loopy Belief propagation on web-spam detection. **(d)** Comparing serializable and non-serializable (racing) execution of the dynamic ALS algorithm in Section 5.8.1 on the Netflix movie recommendation problem. Non-serializable execution exhibits unstable convergence behavior.

4. A range of **schedulers** (Section 5.4.4) which enable rich dynamic computation.
5. Strong **serializability guarantees** (Section 5.4.5) which simplify design, analysis, and implementation of GrAD algorithms and preserves the simple serial execution model.
6. A concurrent **aggregation framework** (Section 5.4.6) which allows users to incrementally estimate global aggregates of the graph.

- An optimized *multicore* implementation (Section 5.5) of the GraphLab abstraction which extends many of the system design techniques developed in Chapter 3 and Chapter 4.
- An optimized *distributed* implementation (Section 5.7) of GraphLab that address distributed data graph representation, mutual exclusion, and fault tolerance.
- GraphLab implementations of a wide range of GrAD algorithms including parameter learning and inference in graphical models, Gibbs sampling as well as several new algorithms including CoEM, Lasso, compressed sensing, image segmentation, and collaborative filtering.
- An experimental evaluation on a range of real-world problems using both the multicore (Section 5.6) and distributed (Section 5.8) implementations demonstrating substantial performance gains over existing systems.

5.2 Common Patterns in MLDM

The GraphLab abstraction explicitly targets algorithms that follow the GrAD methodology and effectively generalizes the approach described in this thesis. In this section review the key properties of the GrAD methodology and discuss how they generalize to a wide range MLDM algorithms.

5.2.1 Sparse Graph Structured Dependencies

Many of the recent advances in MLDM have focused on modeling the *dependencies* between data. By modeling data dependencies, we are able to extract stronger signal from noisy data. For example, modeling the dependencies between similar shoppers and products allows us to make better product recommendations than treating shoppers in isolation. However, directly modeling all possible relationships between dependent data is both computationally intractable and *statistically inefficient*, requiring substantial amounts of data to resolve the form of each relationship. As a consequence, efficient MLDM techniques typically reduce these dependencies to a *sparse graphical structure*. For example, in the earlier chapters we discussed how statistical dependencies can be encoded in the form of sparse probabilistic graphical models.

In this chapter we will consider the more general case in which dependencies between model parameters and data are encoded in the form of a graph. Such graphs include social networks which relate the interests of individuals, term-document co-occurence graphs which connect documents with the key terms they contain, and even physical networks such as those used to represent protein structures and interactions. Identifying and exploiting problem specific sparse structure is a common pattern in MLDM research and we will assume that this graph-structure is known in advance. Alternatively, techniques for resolving the dependency structure typically work by extending upon existing structures.

As we saw in the earlier chapters the algorithms for reasoning about these dependences (e.g., Belief Propagation and Gibbs sampling) operate on *the local neighborhood* of each vertex transforming only a small context of the larger graph. For example, the conditional distribution of each random variable in a large statistical model only depends on its neighbors in the Markov Random Field. Alternatively, the gradient of a parameter in a high-dimensional optimization problem may only depend on a small number of parameters encoded as neighbors in a graph. This explicit locality is essential to enable parallel and distributed computation and efficient data placement. By exploiting latent sparse structure in ML problems, we can develop efficient scalable ML algorithms which are capable expressing rich data dependencies.

While there are MLDM methods which are both data-parallel and model dense data dependencies these methods are not typically scalable to the large data regime and often exhibit poor generalization behavior. For instance learning the complete covariance matrix for a high dimensional data is fully data-parallel procedure which is also unlikely to scale to extremely large problems or yield strong generalization performance. Alternatively, using a sparse structured approximation (e.g., a tree graphical model) can dramatically reduce the computational complexity while improving the generalization performance. By explicitly exploiting sparse dependencies in the design of a parallel framework we obtain *efficient* parallel algorithms that scale.

5.2.2 Asynchronous Iterative Computation

Many important MLDM algorithms iteratively update a large set of parameters. Because of the underlying graph structure, parameter updates (on vertices or edges) depend (through the graph adjacency structure) on the values of other parameters. Furthermore, the presence of cycles in the graph forces most algorithms to search for a fix-point which satisfies all the relationships encoded along the edges. For example, in our work on belief propagation (see Chapter 3) we iteratively updated edge parameters (messages) until the parameters stopped changing by more than some small ϵ. Alternatively, in our work on Gibbs sampling (see Chapter 4), we iterate until we have obtained a sufficient number of samples. In both cases we found that the order in which parameters are updated plays a central role in both the rate of convergence as well as the final fixed-point achieved.

In contrast to **synchronous** systems, which update all parameters simultaneously (in parallel) using parameter values from the previous time step as input, **asynchronous** systems update parameters as resources become available and using the *most recent* parameter values as input. In Chapter 3 we showed how an asynchronous schedule is essential to rapid stable convergence for the belief propagation algorithms. Alternatively, in Chapter 4 we showed how a synchronous approach can lead to statistical divergence and consequently the failure to reach the correct solution.

More generally, by using the most recently computed parameter values and respecting the local *sequential* order asynchronous systems can more efficiently execute a wide range of MLDM algorithms. For example, linear systems (common to many MLDM algorithms) have been shown to converge faster when solved asynchronously [Bertsekas and Tsitsiklis, 1989]. Additionally, there are numerous other cases such as expectation maximization [Neal and Hinton, 1998], and stochastic optimization [Macready et al., 1995, Smola and Narayanamurthy, 2010] where asynchronous procedures have been empirically shown to significantly outperform synchronous procedures. Even classic algorithms such as PageRank can be substantially accelerated when run asynchronous (see Figure 5.1a).

Furthermore from a system design perspective, synchronous computation incurs costly performance penalties since the runtime of each phase is determined by the *slowest* machine. The poor performance of the slowest machine may be caused by a multitude of factors including: load and network imbalances, hardware variability, and multi-tenancy (a principle concern in the Cloud). Even in typical cluster settings, each compute node may also provide other services (e.g., distributed file systems). Imbalances in the utilization of these other services will result in substantial performance penalties if synchronous computation is used.

Serializability

Unlike synchronous computation which has a well defined *deterministic* execution semantics, asynchronous computation can behave non-deterministically especially in the presence of data-dependencies. By ensuring that all parallel executions have a corresponding sequential execution, serializability eliminates many of the challenges associated with designing, implementing, and testing parallel and distributed asynchronous algorithms. Debugging mathematical code in a concurrent program which has data-corruption caused by data races is difficult and time consuming. In addition, many algorithms converge faster if serializability is ensured, and some even require serializability for correctness. For example our work in Chapter 3 exploits serializability along trees to achieve optimality. Alternatively, as discussed in Chapter 4, Gibbs sampling requires serializability to prove statistical correctness. Finally, some algorithms can be made more stable when run with a serializable execution. For instance, the collaborative filtering algorithms such as ALS (Section 5.8.1) can be unstable when allowed to race (Figure 5.1d).

However, serializability often comes at a cost and recently Chafi et al. [2011], Jiayuan et al. [2009] advocated soft-optimization techniques (e.g., allowing computation to intentionally race), but we argue that such techniques do not apply broadly in MLDM. Even for the algorithms evaluated by Chafi et al. [2011], Jiayuan et al. [2009], the conditions under which the soft-optimization techniques work are not well understood and may fail in unexpected ways on different datasets and or cluster configurations.

	Computation Model	**Sparse Depend.**	**Async. Comp.**	**Iterative**	**Prioritized Ordering**	**Enforce Serializ-ability**	**Distrib-uted**
MPI	Messaging	Yes	Yes	Yes	N/A	No	Yes
MapReduce	Par. data-flow	No	No	extensions(**a**)	No	Yes	Yes
Dryad	Par. data-flow	Yes	No	extensions(**b**)	No	Yes	Yes
Pregel/BPGL	GraphBSP	Yes	No	Yes	No	Yes	Yes
Piccolo	Distr. map	No	No	Yes	No	Partially(**c**)	Yes
Pearce et al. [2010]	Graph Visitor	Yes	Yes	Yes	Yes	No	No
GraphLab	**GraphLab**	Yes	Yes	Yes	Yes	Yes	Yes

Table 5.1: **Comparison chart of large-scale computation frameworks.** **(a)** Zaharia et al. [2010] describes and iterative extension of MapReduce. **(b)** Hindman et al. [2009] proposes an iterative extension for Dryad. **(c)** Piccolo does not provide a mechanism to ensure serializability but instead exposes a mechanism for the user to attempt to recover from simultaneous writes. References: MapReduce [Dean and Ghemawat, 2004], Dryad [Isard et al., 2007], Pregel [Malewicz et al., 2010], BPGL [Gregor and Lumsdaine, 2005], Piccolo [Power and Li, 2010]

5.2.3 Dynamic Computation:

In many MLDM algorithms, iterative computation converges asymmetrically. For example, in parameter optimization, often a large number of parameters will quickly converge in a few iterations, while the remaining parameters will converge slowly over many iterations Efron et al. [2004], Elidan et al. [2006]. In Figure 5.1b we plot the distribution of updates required to reach convergence for PageRank. Surprisingly, the majority of the vertices required only a *single* update while only about 3% of the vertices required more than 10 updates. Additionally, prioritizing computation can further accelerate convergence as demonstrated by Zhang et al. [2011] for a variety of graph algorithms including PageRank. If we update all parameters equally often, we waste time recomputing parameters that have effectively converged. Conversely, by focusing early computation on more challenging parameters, we can potentially accelerate convergence. In Figure 5.1c we empirically demonstrate how dynamic scheduling can accelerate convergence of loopy belief propagation (a popular MLDM algorithm). Likewise, in Chapter 4 we showed how prioritization can be used to accelerate the construction of high-likelihood samples.

Unfortunately, the role of parallel schedule for many algorithms is not yet well studied largely because existing tools do not readily support the evaluation of new scheduling techniques. A successful parallel abstraction must provide the ability to support a wide variety of schedules and facilitate schedule comparison.

5.3 High-Level Abstractions

In this section we review several existing high-level abstractions for designing and implementing parallel and distributed algorithms and discuss their limitations in the context GrAD principles. To better organize their presentation we have attempted to divide the abstractions across three general forms of parallelism however in reality individual implementations may blend different forms of parallelism. Furthermore, while we broadly characterize properties of each abstraction is common that instantiations deviate from the general abstraction and potentially address some of the issues we discuss.

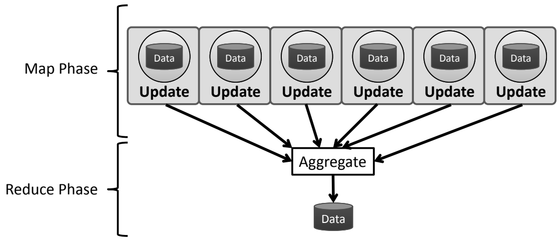


Figure 5.2: A typical MapReduce operation, but drawn using "Update" functions which transforms the Map input data in-place; instead of the canonical data flow model where the Map operation writes its results to some output stream.

5.3.1 Data-Parallel Abstractions

Data parallel abstractions run the same computation on independent sets of data in parallel. Data parallel abstractions are among the most widely used and include frameworks ranging from MapReduce to the general data-flow systems that form the backend modern relational databases. However, data-parallel abstractions typically struggle to efficiently address fixed computational dependencies and the more sophisticated asynchronous dynamics that form the foundation of the GrAD methodology.

Map-Reduce

One of the most widely used high-level distributed programming abstractions is MapReduce. A program implemented in the MapReduce abstraction (Figure 5.2) consists of a `Map` operation and a `Reduce` operation. The `Map` operation is a function which is applied independently and in parallel to each datum (e.g., webpage) in a large data set (e.g., computing the word-count). The `Reduce` operation is an aggregation function which combines the `Map` outputs (e.g., computing the total word count). MapReduce performs optimally only when the algorithm is *embarrassingly parallel* and can be decomposed into a large number of independent computations. The MapReduce framework expresses the class of ML algorithms which fit the Statistical-Query model [Chu et al., 2006] and problems where feature extraction dominates the run-time.

The MapReduce abstraction fails when there are *computational dependencies* in the data. For example, MapReduce can be used to extract features from a massive collection of images but cannot represent computation that depends on small overlapping subsets of images. This critical limitation makes it difficult to represent algorithms that operate on structured models. As a consequence, when confronted with large scale problems, we often abandon rich structured models in favor of overly simplistic methods that are amenable to the MapReduce abstraction. Alternatively, we could try to transform algorithms with computational dependencies into data-parallel variants where computational dependencies have been removed. Transforming MLDM algorithms with computational dependencies into the data-parallel form needed for the map-reduce abstraction can introduce substantial algorithmic inefficiency (e.g., Synchronous BP) and in some cases lead to divergent computation (e.g., Synchronous Gibbs).

While the MapReduce abstraction can be invoked iteratively, it does not provide a mechanism to directly encode iterative computation. Furthermore, to achieve fault-tolerance the MapReduce abstraction must redundantly save the output to a distributed filesystem after each iteration. As a consequence, applying the MapReduce framework to iterative problems with relatively small local computations, a property common to many ML algorithms, can be extremely costly. Recent projects such as Twister [Ekanayake et al., 2010] and Spark [Zaharia et al., 2010] extend MapReduce to the iterative setting and address the costly overhead of disk-based fault-tolerance by introducing novel recovery mechanisms. However, these abstractions are constrained to fully synchronous computation and are limited in their ability to express rich asynchronous and dynamic scheduling.

Dataflow

Dataflow abstractions such as Dryad Isard et al. [2007] and Hyracks Borkar et al. [2011] generalize the Map-Reduce abstraction and represent parallel computation as a directed (typically acyclic) graph with data *flowing* along edges between vertices which correspond to functions which receive information on inbound edges and output results to outbound edges. Dataflow abstractions are ideal for processing streaming data.

While most dataflow abstraction permits rich computational dependencies through sophisticated join operations they are not directly aware of the graph dependency structure in advance. As a consequence data-flow abstractions can easily handle changes in the dependency structure but are not optimized to take advantage of the locality inherent in a static dependency graph. In many cases such join operations may require substantial movement of data. Furthermore, with the exception of recursive dataflow systems such as Datalog [Ceri et al., 1989], most data-flow systems do not directly support iterative or dynamic computation.

5.3.2 Process-Parallel Abstractions

In contrast to data-parallel abstractions, process-parallel abstractions divide computation along separate processes that interact through messages or shared state. These include low level primitives like pthreads and MPI as well as slightly higher-level abstractions based on actor models [Hewitt et al., 1973] like Scala Akka or Erlang [Virding et al., 1996] and distributed key-value models like Piccolo [Power and Li, 2010].

Actor Models

The actor model was first introduced by Hewitt et al. [1973] and encompasses a very broad class of process parallel abstractions. The actor abstraction is generally thought of as fully asynchronous message based abstraction in which processes (actors) communicate by sending and receiving messages and can spawn other actors. There are many implementations of actor models ranging from programming languages such as Erlang to actively developed public APIs such as Akka for Scala.

While the actor model is in many ways well suited to express dynamic asynchronous computation, it does not provide a mechanism to ensure serializability and forces the user to directly manage the movement of information through messages. This makes it difficult to reason about joint state of multiple actors. Furthermore, these abstractions typically do not allow for prioritized computation and instead instantiate actors as processing resources become available. Finally, because the actor model does not impose restrictions on the communication patterns it is unable to isolating load balancing and distributed data

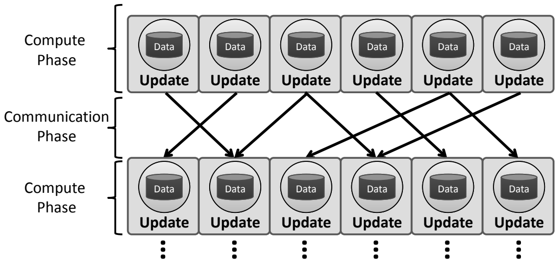


Figure 5.3: The Bulk Synchronous Parallel Model comprises of interleaved fully parallel "compute phases" and "communication phases", where each phase is followed by a global synchronization barrier.

management from the algorithm design forcing the algorithm to address both the ML and systems problem simultaneously.

Piccolo

The Piccolo abstraction, introduced by Power and Li [2010], exposes the program state as a large distributed key-value store and synchronously executes computational kernels in parallel which transform the key-value store. In contrast to the actor abstraction, Piccolo kernels interact through shared state. In addition the Piccolo abstraction exposes a clever commutative associative transaction model that allows for concurrent transactions on a single key-value pair to be combined. However, Piccolo does not explicitly support graph structures and as a consequence locality is achieved through user defined placement functions.

Piccolo implements the Bulk Synchronous Parallel (BSP) model of communication (Figure 5.3), introduced by Valiant [1990], forces computation to be decomposed into small atomic components (like the `Map` in MapReduce), but allows for phased communication between the components and naturally expresses iterative algorithms with computational dependencies. The BSP abstraction divides an algorithm into *super-steps* each consisting of two phases. During the first phase, each processor executes independently (like the Map in Map-Reduce). In the second phase, all processors exchange messages. Finally, a barrier synchronization is performed before the next iteration, ensuring all processors compute and communicate in lockstep. While the BSP model is easier to reason about than the fully asynchronous actors, it cannot efficiently express adaptive asynchronous computation.

5.3.3 Graph-Parallel Abstractions

The need for high-level frameworks that specifically targets graph computation has lead to the emergence of graph-parallel abstractions. A **graph-parallel** abstraction consists of a *sparse* graph $G = \{V, E\}$ and a **vertex-program** Q which is executed in parallel on each vertex $v \in V$ and can interact (e.g., through shared-state or messages) with neighboring instances $Q(u)$ where $(u, v) \in E$. In contrast to more general message passing and actor models models, graph-parallel abstractions constrain the interaction of vertex-program to a graph structure enabling the optimization of data-layout and communication.

Pregel

Malewicz et al. [2010] introduced the Pregel bulk synchronous *message passing* abstraction in which all vertex-programs run simultaneously in a sequence of super-steps. Within a **super-step** each program instance $Q(v)$ receives all messages from the previous super-step and sends messages to its neighbors in the next super-step. A barrier is imposed between super-steps to ensure that all program instances finish processing messages from the previous super-step before proceeding to the next. The program terminates when there are no messages remaining and every program has voted to halt. Pregel introduces commutative associative **message combiners** which are user defined functions that merge messages destined to the same vertex. These message combiners behave similarly to the commutative associative transactions within the Piccolo abstraction. In many ways the Pregel abstraction is a generalization of the classic Systolic array[Shapiro, 1988]. While systolic arrays focused on a fixed often physical network topology with processors at each node, Pregel virtualizes each processor as vertex-program and allows for arbitrary graph topologies.

The following is an example of the PageRank vertex-program implemented in Pregel. The vertex-program receives the single incoming message (after the combiner) which contains the sum of the PageRanks of all in-neighbors. The new PageRank is then computed and sent to its out-neighbors.

```
Message combiner(Message m1, Message m2) :
 return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
 float total = msg.value();
 vertex.val = 0.15 + 0.85*total;
 foreach(nbr in out_neighbors) :
   SendMsg(nbr, vertex.val/num_out_nbrs);
```

One of the key limitations of the Pregel abstraction is that the user must architect the movement of program state between vertex-programs through messages. For example, if we consider the above PageRank example we see that each vertex generates a *separate but identical* message which is sent to each of its neighbors. If each of its neighbors is physically mapped to a separate machine this might be reasonable. However, if many of its neighbors are on the same machine this could lead to *substantial* additional network overhead. We will return to the limitations of messaging abstraction throughout this and the subsequent chapter.

While Pregel improves upon Piccolo by introducing the graph structure it retains the synchronous execution semantics. As a consequence, we can naturally express Synchronous BP in Pregel but we cannot to express the more asynchronous variants of BP. Moreover, the Pregel abstraction is not designed for dynamic computation and specialized structured schedules which are central to the GrAD methodology. As we have already demonstrated, by skipping unnecessary vertex-programs and by executing vertex-programs along spanning trees, we can reduce wasted computation and move information across the graph more efficiently.

5.4 The GraphLab Abstraction

At a high-level, GraphLab is an *asynchronous distributed shared-memory graph-parallel* abstraction in which vertex programs have shared access to information on adjacent edges and vertices and can schedule neighboring vertex-programs to be executed in the future. By targeting GrAD methodology, GraphLab achieves a balance between low-level and high-level abstractions. Unlike many low-level abstractions (e.g., MPI, PThreads), GraphLab insulates users from the complexities of synchronization, data races and deadlocks by providing a high level data representation through the *data graph* and automatically maintained *serializability* guarantees through configurable *exclusion policies*. Unlike many high-level abstractions (i.e., MapReduce), GraphLab can express complex computational dependencies using the data graph and provides sophisticated *scheduling primitives* which can express iterative parallel algorithms with dynamic scheduling.

The GraphLab abstraction consists of three main parts, the data graph, the vertex-program, and the sync operation. The data graph (Section 5.4.1) represents user modifiable program state, and stores both the mutable user-defined data and encodes the sparse computational dependencies. The vertex-program (Section 5.4.2) represents the user computation and operate on the data graph by transforming data in small overlapping contexts called scopes. Finally, the sync operation (Section 5.4.6) concurrently maintains global aggregates. To ground the GraphLab abstraction in a concrete problem, we will use the PageRank algorithm Page et al. [1999] as a running example.

Example 5.4.1 (PageRank)**.** *The PageRank algorithm recursively defines the rank of a webpage v:*

$$\boldsymbol{R}(v) = \frac{\gamma}{n} + (1-\gamma) \sum_{u \text{ links to } v} w_{u,v} \times \boldsymbol{R}(u) \tag{5.1}$$

in terms of the weighted $w_{u,v}$ ranks $\boldsymbol{R}(u)$ of the pages u that link to v as well as some probability γ of randomly jumping to a new page (typically $\gamma = 0.15$). The PageRank algorithm iterates Eq. (5.1) until the PageRank changes by less than some small value ϵ.

5.4.1 Data Graph

The GraphLab abstraction stores the program state as a directed graph called the **data graph**. The data graph $G = (V, E, D)$ is a container that manages the user defined data D. Here we use the term *data* broadly to refer to model parameters, algorithm state, and even statistical data. Users can associate arbitrary data with each vertex $\{D_v : v \in V\}$ and edge $\{D_{u \to v} : \{u, v\} \in E\}$ in the graph. However, as the GraphLab abstraction does not depend on edge directions, we also use $D_{u \leftrightarrow v}$ to denote the data on both edge directions $u \to v$ and $v \to u$. Since some MLDM applications require directed edge data (e.g., weights on directed links in a web-graph), we provide the ability to store and retrieve data associated with each direction along an edge. Finally, while the graph data is mutable, the structure is *static* and cannot be changed during execution.

Example 5.4.2 (PageRank: Example 5.4.1)**.** *The data graph is directly obtained from the web graph, where each vertex corresponds to a web page and each edge represents a link. The vertex data D_v stores $\boldsymbol{R}(v)$, the current estimate of the PageRank, and the edge data $D_{u \to v}$ stores $w_{u,v}$, the directed weight of the link.*

The data graph naturally represents the state of a wide range of machine learning algorithms. For example in belief propagation, the data graph is the pairwise MRF, with the vertex data D_v to storing the node potentials and the directed edge data $D_{u \to v}$ storing the BP messages. If the MRF is sparse then the data graph is also sparse and GraphLab will achieve a high degree of parallelism. The data graph can also be used to represent many algorithms that do not explicitly operate on graph structures. For example, the shooting algorithm (coordinate-wise sub-gradient descent) for the Lasso can be transformed into a data graph by connecting each weight with examples that have non-zero feature values for that weight.

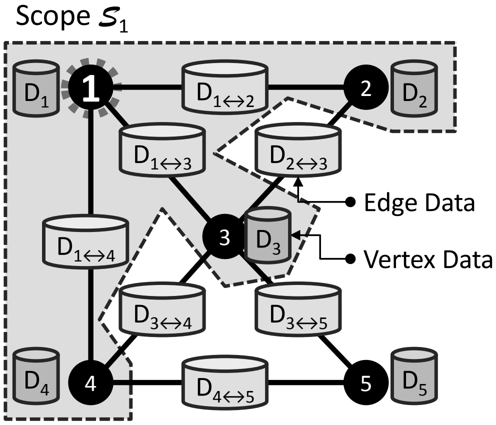


Figure 5.4: The **data graph** and scope $\mathcal{S}_1$. Gray cylinders represent the user defined vertex and edge data while the irregular region containing the vertices $\{1, 2, 3, 4\}$ is the **scope**, $\mathcal{S}_1$ of vertex 1. An **vertex program** applied to vertex 1 is able to read and modify all the data in $\mathcal{S}_1$ (vertex data D_1, D_2, D_3, and D_4 and edge data $D_{1\leftrightarrow 2}$, $D_{1\leftrightarrow 3}$, and $D_{1\leftrightarrow 4}$).

As we will show in subsequent sections, the explicit data graph representation of the program state also enables the GraphLab runtime to effectively place the entire program state in a distributed environment, automatically enforce powerful serializability models, and even achieve efficient fault-tolerance.

5.4.2 GraphLab Vertex Programs

Computation is encoded in the GraphLab abstraction in the form of a user defined vertex program. A GraphLab **vertex program** is a stateless procedure that modifies the data within the scope of a vertex and schedules the future execution of vertex programs on neighboring vertices. The **scope** of vertex v (denoted by $\mathcal{S}_v$) is the data stored on v, as well as the data stored on all adjacent vertices and adjacent edges (Figure 5.4). A GraphLab vertex program is analogous to the map function in MapReduce, but unlike in MapReduce, vertex programs are permitted to access and modify *overlapping* contexts in the graph.

A GraphLab vertex program takes as an input a vertex v and its scope $\mathcal{S}_v$ and returns the new versions of the data in the scope as well as a set vertices $\mathcal{T}$:

$$\textbf{Vprog} : f(v, \mathcal{S}_v) \to (\mathcal{S}_v, \mathcal{T})$$

After executing a vertex program the modified data in $\mathcal{S}_v$ is written back to the data graph. We refer to the neighborhood $\mathcal{S}_v$ as the scope of v because $\mathcal{S}_v$ defines the extent of the graph that can be accessed by the vertex program f when applied to v. The set of vertices $\mathcal{T}$ are *eventually* executed by applying the vertex program $f(u, \mathcal{S}_u)$ for all $u \in \mathcal{T}$ following the execution semantics described later in Section 5.4.3.

Rather than adopting a message passing (e.g., Pregel) or data flow (e.g., Dryad) model, GraphLab allows the user defined vertex programs complete freedom to read and modify any of the data on adjacent vertices and edges. This simplifies user code and eliminates the need for the users to orchestrate the movement of data. This allows the implementation of the system to choose when and how to move vertex and edge data in a distributed environment. By controlling what vertices are returned in $\mathcal{T}$ and thus are eventually executed, GraphLab vertex programs can efficiently express adaptive computation. For example, a vertex program may choose to return (schedule) its neighbors only when it has made a substantial change to its local data.

There is an important difference between Pregel and GraphLab in how dynamic computation is expressed. GraphLab decouples the scheduling of future computation from the movement of data. As a consequence, GraphLab vertex programs have access to data on adjacent vertices even if the adjacent vertices did not schedule the current execution. Conversely, Pregel vertex programs are initiated by messages and can only access the data in the messages sent during the previous super-step. This restriction limits what can be expressed and explicitly couples program dynamics to the user orchestrated movement of data.

For instance, dynamic PageRank is difficult to express in Pregel since the PageRank computation for a given page requires the PageRank values of all adjacent pages even if some of the adjacent pages *have not recently changed*. Therefore, the decision to send data (PageRank values) to neighboring vertices cannot be made by the sending vertex (as required by Pregel) but instead must be made by the receiving vertex. GraphLab, naturally expresses the *pull* behavior, since adjacent vertices are only responsible for *scheduling*, and vertex programs can always *directly read* adjacent vertex values even if they have not changed.

Example 5.4.3 (PageRank: Example 5.4.1)**.** *The vertex program for PageRank (defined in Alg. 5.1) computes a weighted sum of the current ranks of neighboring vertices and assigns it as the rank of the current vertex. The algorithm is adaptive: neighbors are scheduled for execution only if the value of current vertex changes by more than a predefined threshold.*

Algorithm 5.1: PageRank Vertex Program in the GraphLab Abstraction

```
Input: Vertex data $\mathbf{R}(v)$ from $\mathcal{S}_v$
Input: Edge data $\{w_{u,v} : u \in \mathcal{N}_v\}$ from $\mathcal{S}_v$
Input: Neighbor vertex data $\{\mathbf{R}(u) : u \in \mathcal{N}_v\}$ from $\mathcal{S}_v$
$\mathbf{R}_{\text{old}}(v) \leftarrow \mathbf{R}(v)$                    // Save old PageRank
$\mathbf{R}(v) \leftarrow \gamma / n$
foreach $u \in \mathcal{N}_v$ do                                      // Loop over neighbors
    $\mathbf{R}(v) \leftarrow \mathbf{R}(v) + (1 - \gamma) * w_{u,v} * \mathbf{R}(u)$
// If the PageRank changes sufficiently
if $|\boldsymbol{R}(v) - \boldsymbol{R}_{old}(v)| > \epsilon$ then
    // Schedule neighbors to be executed
    return $\{u : u \in \mathcal{N}_v\}$
Output: Modified scope $\mathcal{S}_v$ with new $\mathbf{R}(v)$
```

Algorithms that follow the GrAD principles can naturally be expressed as GraphLab vertex programs. For example, probabilistic inference algorithms discussed in earlier chapters can directly translated into vertex programs. Like wise other classical algorithms such as expectation propagation Minka [2001] and mean field variational inference Xing et al. [2003] can all be expressed using vertex programs which read the current assignments to the parameter estimates on neighboring vertices and edges and then apply sampling or optimization techniques to update parameters on the local vertex.

5.4.3 The GraphLab Execution Model

The GraphLab execution model, presented in Alg. 5.2, follows a simple single loop semantics. The input to the GraphLab abstraction consists of the data graph $G = (V, E, D)$, a vertex program, and an initial *active set* of vertices $\mathcal{T}$ to be executed. While there are vertices remaining in $\mathcal{T}$, the algorithm removes (Line 1)

Algorithm 5.2: GraphLab Execution Model

```
  Input: Data Graph $G = (V, E, D)$
  Input: Initial vertex set $\mathcal{T} = \{v_1, v_2, ...\}$
  while $\mathcal{T}$ is not Empty do
1     $v \leftarrow$ RemoveNext($\mathcal{T}$)
2     $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$
3     $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$
  Output: Modified Data Graph $G = (V, E, D')$
```

and executes (Line 2) vertices, adding any new vertices back into $\mathcal{T}$ (Line 3). The active set $\mathcal{T}$ behaves like a classic set and therefore contains at most one instance of each vertex. As a consequence if a vertex is added more than once before being executed, then it is executed only once. The resulting data graph is returned to the user on completion.

5.4.4 Scheduling Vertex Programs

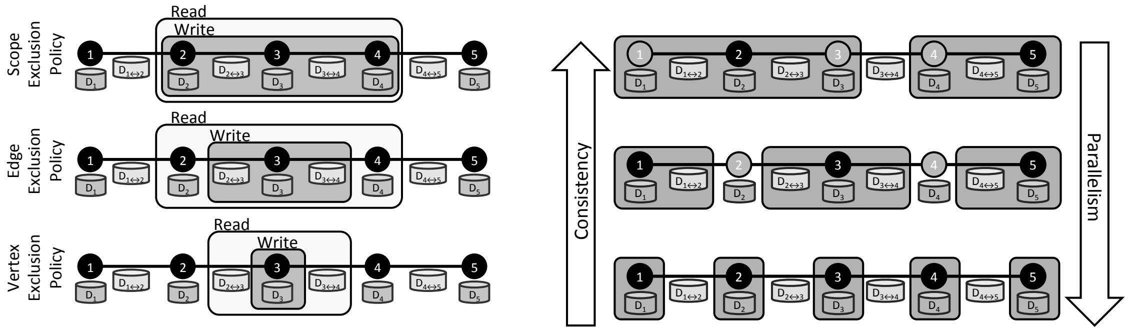
The order in which vertices are returned from `RemoveNext`($\mathcal{T}$) (Line 1) is determined by the GraphLab **scheduler**. GraphLab provides a collection of schedulers each with different behaviors. While all schedulers must ensure that all vertices in the active set $\mathcal{T}$ are eventually executed, each scheduler is free to choose an order that optimizes system performance, parallelization, and algorithm convergence.

The GraphLab framework provides three fundamental schedulers which are capable of expressing a wide range of computation. The simplest of these schedulers is the **sweep** scheduler which always executes vertices in $\mathcal{T}$ in a canonical order. Typically the canonical order is either a random permutation of the vertices or a specifically chosen order based on properties of the vertices or the structure of the graph. The sweep scheduler is ideally suited for algorithms like Gibbs sampling which repeatedly iterate over all vertices.

As we observed in our earlier work in graphical model inference, it is often advantageous to schedule computation along a wavefront. This can be accomplished by using the **FiFo** scheduler which executes vertices in the order in which they are added. The FiFo scheduler preserves the set semantics of $\mathcal{T}$ and so repeated additions of the same vertex are ignored until the vertex leaves the FiFo schedule. As a consequence the FiFo schedule effectively maintains a *frontier* set and enables a bread-first traversal of the graph.

While the presence of the active set $\mathcal{T}$ of vertices automatically leads to *adaptive* computation, we demonstrated in our earlier work (Chapter 3 and Chapter 4) that dynamic *prioritization* can further accelerate the convergence. Therefore GraphLab provides a **priority** scheduler which associates a priority with each vertex and returns the active vertices in order of priority. If a vertex is added more than once than its highest priority value is used and so vertex programs can promote the priority of active vertices. The priority scheduler is used to implement algorithms like residual belief propagation.

In addition to the three basic schedules, we also have implemented several special purpose schedulers. For example, motivated by our work on the Splash BP and Splash Gibbs sampling algorithms we developed a specialized Splash scheduler which automatically constructs bounded work Splashes. We believe that work in improved scheduling heuristics can lead to more efficient parallel algorithms. As a consequence GraphLab makes it easy for users to compose different schedulers with the same vertex program.



(a) Exclusion Policies

(b) Serializability and Parallelism

Figure 5.5: **(a)** The read and write permissions for a vertex program executed on vertex 3 under each of the exclusion policies. Under the **scope exclusion policy** the vertex program has complete read-write access to its entire scope. Under the **edge exclusion policy**, the vertex program has only read access to adjacent vertices. Finally, **vertex exclusion policy** only provides write access to the central vertex data. **(b)** The trade-off between serializability and parallelism. The dark rectangles denote the write-locked regions that cannot overlap. Vertex program are executed on the dark vertices concurrently. Under the stronger serializability models fewer functions can run concurrently.

When executed in a distributed environment each scheduler can optionally relax its scheduling guarantees to improve system performance. For example, in the distributed setting each scheduler only maintain its scheduling invariant with respect to vertices explicitly assigned to that machine. However, regardless of the parallel or distributed execution each scheduler must always maintain the invariant that all schedule vertices are executed eventually.

5.4.5 Ensuring Serializable Execution

The GraphLab abstraction presents an inherently *sequential* model of computation which is automatically translated into a *parallel execution* by allowing multiple processors to execute the same loop on the same graph, removing and executing different vertices concurrently. To retain the sequential execution semantics we prevent overlapping computation from running concurrently. However, the extent to which computation can *safely* overlap depends on the user defined computation within the vertex program as well as the stability of the underlying algorithm. We therefore introduce several **exclusion policies** that allow the runtime to optimize the parallel execution while maintaining serializability.

The GraphLab runtime ensures a **serializable** execution. A serializable execution implies that there exists a corresponding serial schedule of vertex programs that when executed *sequentially* following Alg. 5.2 produces the same values in the data-graph.

Definition 5.4.1 (GraphLab Serializability Guarantee)**.** *For every parallel execution of the GraphLab abstraction, there exists a sequential ordering on the vertices such that vertex programs executed sequentially according to that ordering produce the same data graph.*

By ensuring serializability, GraphLab simplifies reasoning about highly-asynchronous dynamic computation in the distributed setting. For example, in a serializable execution it is not possible for adjacent vertices to run concurrently. As a consequence, if an invariant over the neighborhood of a vertex is true at the start of the vertex program it will remain true until after the vertex program completes. Alternatively, a non-serializable execution would make it extremely difficult to maintain neighborhood invariants.

Serializability is achieved through three different mutual exclusion policies which prevent vertex programs with overlapping state from executing concurrently (see Figure 5.5a). The strongest policy is the **scope exclusion** policy which prevents vertex programs with overlapping scopes from executing concurrently. As a consequence the scope exclusion policy ensures that during the execution of the vertex program on the vertex v no other function will read or modify data within $\mathcal{S}_v$. However, the scope exclusion policy limits the potential parallelism since concurrently executing vertices cannot share any common neighbors (i.e., must be at least two vertices apart) as illustrated in Figure 5.5b.

The vast majority of vertex programs only *read* the values of adjacent vertices. In this case we can permit vertices with common neighbors to run concurrently and *still ensure serializability*. For instance, the PageRank vertex program only requires read access to neighboring vertices. To provide greater parallelism while retaining serializability, GraphLab defines the **edge exclusion** policy. The edge exclusion policy ensures each vertex program has exclusive read-write access to its vertex and adjacent edges but read only access to adjacent vertices as illustrated in Figure 5.5a). As a consequence, the edge exclusion policy increases parallelism by allowing vertex programs with slightly overlapping scopes to safely run in parallel (see Figure 5.5b).

The **vertex exclusion** policy is the weakest of the three policies and guarantees a serializable execution only if vertex programs do not read or write to neighboring vertices and do not write to adjacent edges. Under the vertex exclusion policy neighboring vertex programs can run concurrently. By allowing neighboring vertex programs to run, the vertex exclusion policy allows maximum parallelism. While very few vertex programs satisfy this weak policy, it can be useful when initializing data and more importantly to explore the non-serializable executions of vertex programs.

We summarize the guarantees of each of the three mutual exclusion policies in the following:

Proposition 5.4.4. *GraphLab guarantees serializability under the following three conditions:*

1. *The **scope exclusion** policy is used*
2. *The **edge exclusion** policy is used and vertex programs do not modify data on adjacent vertices.*
3. *The **vertex exclusion** policy is used and vertex programs do not read or write to adjacent vertices and do not write to adjacent edges.*

Because mutual exclusion and data serializability are maintained by the GraphLab abstraction, the user does not have to deal with the challenges of avoiding deadlock or data races. This is an important property since, preventing deadlock and race conditions is one of the principle challenges of parallel and distributed algorithm design. Moreover, if the user selects the correct mutual exclusion policy and ensures that every vertex program returns, the GraphLab abstraction guarantees a race free and deadlock free execution.

The serializability property of the GraphLab abstraction also enables the expression and analysis of a much richer class of vertex programs. For example, we can easily implement graph coloring by having each vertex choose a color that is different from its neighbors. We can prove that under the edge exclusion policy, the graph coloring vertex program will converge and each vertex will be executed only once. In contrast, under the bulk synchronous parallel model adopted by abstractions like Pregel, the same algorithm will never converge. Furthermore, we can directly apply many of the theoretical tools for analyzing serial graph algorithms to the analysis of parallel graph algorithms running in the GraphLab abstraction. For example, by running a Gibbs sampler vertex program under the edge exclusion policy, we can directly apply existing theoretical proofs of ergodicity to the parallel execution.

While a serializable execution is essential when designing, implementing, and debugging complex MLDM algorithms, some algorithms are robust to non-serializable executions and may benefit from the increased parallelism achieved by running under a weaker mutual exclusion policy. For example, work by Chafi et al. [2011], Recht et al. [2011] has demonstrated that some algorithms can run efficiently even in a fully asynchronous setting. Therefore we allow users to choose a weaker mutual exclusion policy.

5.4.6 Aggregation Framework

In many MLDM algorithms it is necessary to maintain global statistics describing data stored in the data graph. For example, many statistical inference algorithms require tracking global convergence estimators. To address this need, the GraphLab abstraction defines global values that may be read by vertex programs, but are written using aggregators. Similar to aggregates in Pregel, A GraphLab **global aggregate** is an commutative associative sum:

$$Z = \textbf{Finalize}\left(\bigoplus_{v \in V} \textbf{Map}(\mathcal{S}_v)\right) \tag{5.2}$$

defined over all the scopes in the graph. Unlike Pregel, GraphLab aggregates introduce a finalization phase, $\textbf{Finalize}(\cdot)$, to support tasks, like normalization, which are common in MLDM algorithms. Also in contrast to Pregel, where the aggregation runs after each super-step, aggregation in GraphLab runs continuously in the background to maintain updated estimates of the global value.

Since every vertex program can access global values, ensuring serializability of global aggregates with respect to vertex programs is costly and requires synchronizing and blocking all vertex programs while the global value is computed. Therefore just as GraphLab has multiple exclusion policies for vertex programs, GraphLab similarly provides the option to enable or disable serializability of global aggregate computations.

Example 5.4.5 (PageRank: Example 5.4.1)**.** *We can compute the second most popular page on the web by defining the following aggregate operation:*

$$\textbf{\textit{Map}}: \quad (\mathcal{S}_v) \rightarrow (\boldsymbol{R}(v), -\infty) \tag{5.3}$$

$$\oplus: \quad (\boldsymbol{R}(a_1), \boldsymbol{R}(a_2)), (\boldsymbol{R}(b_1), \boldsymbol{R}(b_2)) \rightarrow \textit{TopTwo}\left(\{\boldsymbol{R}(a_1), \boldsymbol{R}(a_2), \boldsymbol{R}(b_1), \boldsymbol{R}(b_2)\}\right) \tag{5.4}$$

$$\textbf{\textit{Finalize}}: \quad (\boldsymbol{R}(a_1), \boldsymbol{R}(a_2)) \rightarrow \boldsymbol{R}(a_2) \tag{5.5}$$

We may want to update the global estimate every $\tau = |V|$ vertex updates.

5.5 Multicore Design

We first implemented an optimized version of the GraphLab framework for the shared-memory multicore setting in the form of a C++ API. The user first constructs and populates the data graph with initial data. The user also constructs a GraphLab engine specifying the vertex program, mutual exclusion policy, and scheduler as well as any aggregation operations. Computation is initiated by passing the data graph and an initial set of vertices to the engine. The engine worker threads then run concurrently until no active vertices remain and the data graph is modified in place. When the engine returns the data graph has been updated and there are no remaining active vertices.

We implemented the C++ API by building upon our work in parallel inference in graphical models. The data graph was implemented as a simple generic container with user defined objects associated with each vertex and edge based on the graph data structures used to represent graphical models.

Likewise, the mutual exclusion policies were implemented by extending the ordered locking used in the Markov blanket locking protocol discussed in Section 4.4. We associate a readers-writer lock with each vertex. The different exclusion policys are implemented using different locking protocols. Vertex exclusion is achieved by acquiring a write-lock on the central vertex of each requested scope. Edge exclusion is achieved by acquiring a write lock on the central vertex, and read locks on adjacent vertices. Finally, scope exclusion is achieved by acquiring write locks on the central vertex and all adjacent vertices. As with the Markov blanket locking protocol, deadlocks are avoided by acquiring locks sequentially following a canonical order.

The primary challenge was designing efficient parallel schedulers and this was achieved using the same techniques discussed in Chapter 3. To attain maximum performance we had to address issues related to parallel memory allocation, concurrent random number generation, and cache efficiency. Since mutex collisions can be costly, lock-free data structures and atomic operations were used whenever possible. To achieve the same level of performance for parallel learning system, the MLDM community would have to repeatedly overcome many of the same *time consuming* system design challenges needed to build GraphLab.

5.6 Multicore GraphLab Evaluation

To demonstrate the expressiveness of the GraphLab abstraction and illustrate the parallel performance gains it provides, we implemented four popular ML algorithms and evaluate these algorithms on large real-world problems using a 16-core computer with 4 AMD Opteron 8384 processors and 64GB of RAM. All single core (serial) timing was run with locking disabled to eliminate as much system overhead as possible. We fount that often the system optimizations made to GraphLab resulted in better performance than our previous application specific implementations of many of the algorithms.

5.6.1 Parameter Learning and Inference in a Markov Random Field

To demonstrate how the various components of the GraphLab framework can be assembled to build a complete “pipeline” for probabilistic reasoning we use GraphLab to solve a novel three-dimensional retinal image restoration task. In this task we begin with raw three-dimensional laser density estimates and then use GraphLab to generate composite statistics, learn parameters for a large three-dimensional grid pairwise MRF, and then finally compute expectations for each voxel using Loopy BP. Each of these tasks requires both local iterative computation and global aggregation as well as several different computation schedules.

We begin by using the GraphLab data graph to build a large (256x64x64) three dimensional MRF in which each vertex corresponds to a voxel in the original image. We connect neighboring voxels in the 6 axis aligned directions. We store the density observations and beliefs in the vertex data and the BP messages in

Algorithm 5.3: BP Vertex Program

```
Compute the local belief $b(x_v)$ using $\{D_{*\to v} D_v\}$
active neighbors $\mathcal{T}$
foreach $(v \to t) \in (v \to *)$ do
    Update $m_{v\to t}(x_t)$ using $\{D_{*\to v}, D_v\}$
    residual $\leftarrow \left|\left| m_{v\to t}(x_t) - m^{\text{old}}_{v\to t}(x_t) \right|\right|_1$
    if residual $>$ Termination Bound then
        Add neighbor to active neighbors $\mathcal{T} \leftarrow \mathcal{T} \cup (t, \text{residual})$ with residual priority
    return active neighbors $\mathcal{T}$
```

Algorithm 5.4: Parameter Learning Aggregator

```
Map: return deviation from empirical marginal for each pixel
$\oplus$ : Combine image statistics by averaging
Finalize: Apply gradient step to $\lambda$ using the final accumulator
```

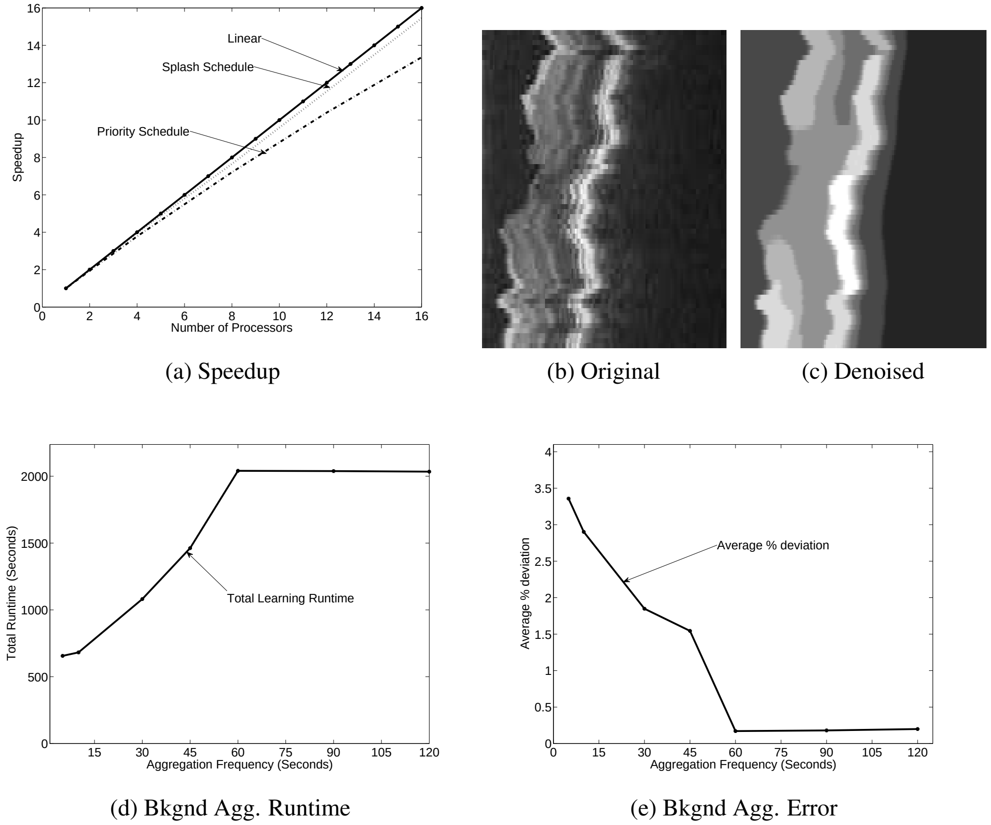
the directed edge data. We will use the aggregation framework to maintain three global parameters which determine the smoothing in each dimension along the MRF. Prior to learning the model parameters, we first use the GraphLab aggregation framework to compute axis-aligned averages as a proxy for ground-truth smoothed images along each dimension. We then performed concurrent *learning* and *inference* in GraphLab by using the aggregation framework (Alg. 5.4) to aggregate inferred model statistics and apply a gradient descent procedure. To the best of our knowledge, this is the first time graphical model parameter learning and BP inference have been done concurrently.

Results: In Figure 5.6a we plot the speedup of the parameter learning algorithm, executing inference and learning sequentially. We found that the Splash BP scheduler outperforms other scheduling techniques enabling a factor 15 speedup on 16 cores. We then evaluated concurrent parameter learning and inference by allowing the aggregation framework to run concurrently with inference (Figure 5.6d and Figure 5.6e). In Figure 5.6b and Figure 5.6c we plot examples of noisy and denoised cross sections respectively.

By running a background aggregation operation at the right frequency, we found that we can further accelerate parameter learning while only marginally affecting the learned parameters. In fact, by frequently taking gradient steps (once every 3 seconds), before belief propagation has converged to an accurate gradient estimator, we are able to obtain a *factor of three* reduction in the overall running time. The substantial reduction has leads to slightly different parameters (differs by 4%) but does not affect the quality of the final prediction. We believe this provides a strong indication of the potential of the asynchronous approach in the GrAD methodology.

5.6.2 Gibbs Sampling

To demonstrate the power of the GraphLab abstraction we implement a series of provably ergodic (asymptotically consistent) Gibbs samplers. In Chapter 4 we demonstrated how the Gibbs sampling algorithm is inherently sequential and has frustrated efforts to build asymptotically consistent parallel samplers. However, by exploiting serializability we are able to easily construct an asymptotically consistent parallel



(a) Speedup

(b) Original

(c) Denoised

(d) Bkgnd Agg. Runtime

(e) Bkgnd Agg. Error

Figure 5.6: *Retinal Scan restoration* **(a)** Speedup relative to the best single processor runtime of parameter learning using priority, and Splash schedules. **(b,c)** A slice of the original noisy image and the corresponding expected pixel values after parameter learning and restoration. **(d)** The total runtime in seconds of parameter learning and **(e)** the average percent deviation in learned parameters plotted against the time between gradient steps using the Splash schedule on 16 processors.

Gibbs sampler. In this example we compare the FiFo and Sweep schedulers with a custom built chromatic schedule from Section 4.3.

To apply the chromatic schedule we needed to first construct a coloring of the data graph. This is accomplished by implementing the standard greedy graph coloring algorithm as a GraphLab vertex program which examines the colors of the neighboring vertices of v, and sets v to the first unused color. We use the edge exclusion policy to ensure that the parallel execution retains the same guarantees as the sequential version. We were able to color the graph in fractions of a second. Once we have a valid graph coloring we can then revert to the *vertex exclusion* policy when using the Chromatic schedule.

To evaluate the GraphLab implementation of the parallel Gibbs sampler we consider the challenging task of marginal estimation on a factor graph representing a protein-protein interaction network obtained from Elidan et al. [2006] by generating 10,000 samples. The resulting graph (MRF) has roughly 100K edges and 14K vertices. Unlike the earlier pairwise MRFs these networks are generally more challenging with higher order factors and more irregular structure and substantially greater connectivity. In Figure 5.7 we plot the color distribution for the data-graph and we see that their are relatively few colors. As a baseline for comparison we also ran the GraphLab version Splash Loopy BP (Chapter 3) algorithm.

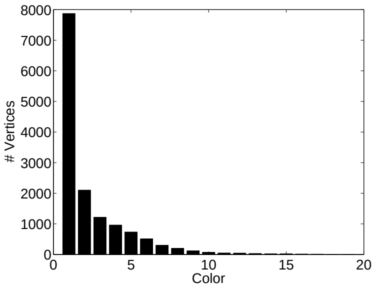


Figure 5.7: Color distribution for the protein-protein interaction network.

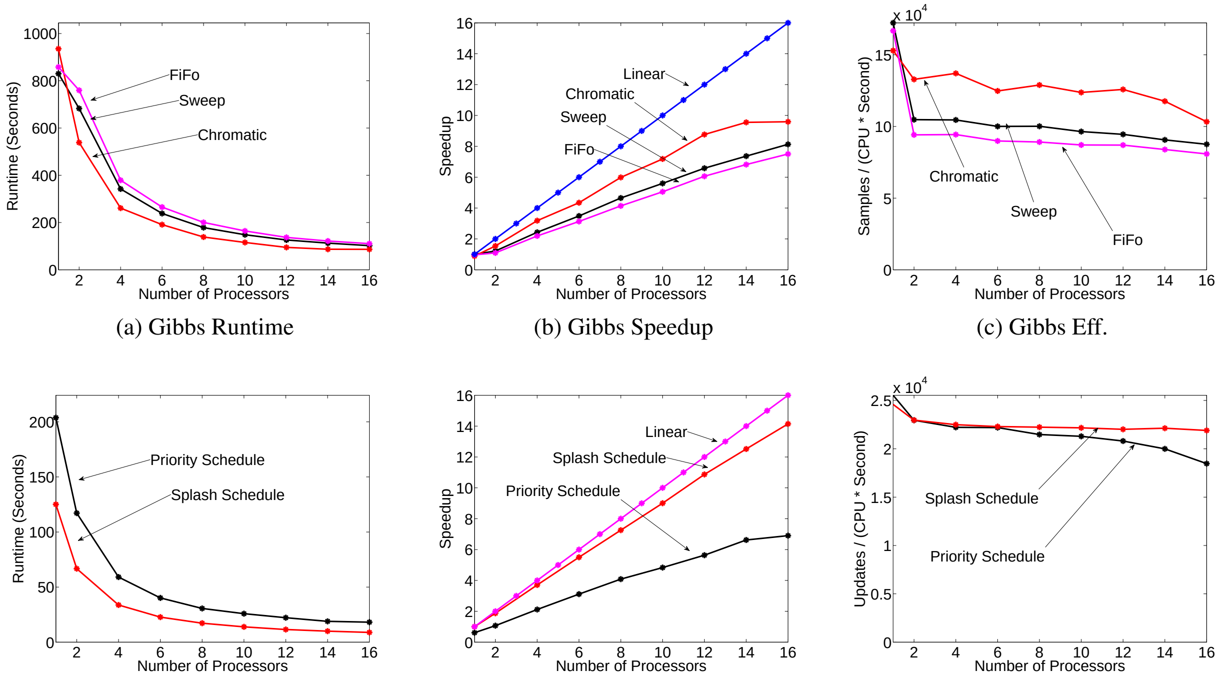
Results: In Figure 5.8 we present the runtime, speedup, and efficiency results for Gibbs sampling and Loopy BP. Consistent with our earlier findings in Chapter 4 the chromatic sampler is able to scale slightly better because we can eliminate the need for the heavy locking in the edge exclusion policy. In contrast the Loopy BP speedup demonstrates considerably better scaling with factor of 15 speedup on 16 processor. Also consistent with our earlier findings the SplashBP scheduler is able to accelerate convergence. The larger cost per BP update in conjunction with the ability to run a fully asynchronous schedule enables Loopy BP to achieve relatively uniform update efficiency compared to Gibbs sampling. It is worth noting that the difference between the SplashBP and ResidualBP algorithms in the GraphLab framework is a single command line argument (i.e., `--scheduler=splash` vs. `--scheduler=priority`).

5.6.3 Named Entity Recognition (NER)

To illustrate how GraphLab scales in settings with large structured models we designed and implemented a parallel version of CoEM [Jones, 2005, Nigam and Ghani, 2000], a semi-supervised learning algorithm for named entity recognition. Named Entity Recognition (NER) is the task of determining the type (e.g., *Person*, *Place*, or *Thing*) of a **noun-phrase** (e.g., *Obama*, *Chicago*, or *Car*) from its **context** (e.g., *"President __"*, *"lives near __."*, or *"bought a __."*). NER is used in many natural language processing applications as well as information retrieval. In this application we obtained a large crawl of the web from the NELL project [Carlson et al., 2010] and constructed two NER datasets (Table 5.2) in which we counted the number of occurrences of each noun-phrase in each context. Starting with a small seed set of pre-labeled noun-phrases, the CoEM algorithm labels the remaining noun-phrases and contexts (see Table 5.2b) by alternating between estimating the type of each noun-phrase given the types of its contexts and estimating the type of each context given the types of its noun-phrases.

The data graph for the NER problem is bipartite with one set of vertices corresponding to noun-phrases and other corresponding to each context. There is an edge between a noun-phrase and a context if the noun-phrase occurs in the context. The vertices store the estimated distribution over types and the edges store the number of times the noun-phrase appears in a context. The CoEM vertex program (Alg. 5.5) simply updates the estimated distribution for a vertex (either noun-phrase or context) based on the current distribution for neighboring vertices. We implemented a dynamic version of the CoEM algorithm in which adjacent vertices are rescheduled if the type distributions at a vertex changes by more than some predefined threshold (10^{-5}). The CoEM vertex program is relatively fast, requiring only a few floating point operations, and therefore stresses the GraphLab implementation by requiring GraphLab to manage massive amounts of fine-grained parallelism.

We plot in Figure 5.9a and Figure 5.9b the speedup on both small and large datasets respectively. On both



(a) Gibbs Runtime

(b) Gibbs Speedup

(c) Gibbs Eff.

(d) BP Runtime

(e) BP Speedup

(f) BP Eff.

Figure 5.8: *MRF Inference* We ran the Gibbs sampler vertex program with three different schedules and report the runtime **(a)**, speedup **(b)**, and work efficiency **(c)**. Consistent with the results in Chapter 4 we find that the chromatic schedule is the fastest and most efficient in terms of raw sample generation rate. To provide a basis for comparison we also ran loopy belief propagation on the same problem evaluating two different schedulers and again reporting runtime **(d)**, speedup **(e)**, and work efficiency **(f)**.

datasets we see relatively strong scaling performance. However, we see better scaling on the larger dataset which is expected since the computation time for each vertex program goes up with the number of types hiding much of the GraphLab system overhead. With 16 parallel processors, we could complete three full Round-robin iterations on the large dataset in less than 30 minutes (including data loading time). As a comparison, a comparable Hadoop implementation[1] took approximately 7.5 hours to complete the exact same task, executing on an average of 95 cpus.

Using the flexibility of the GraphLab framework we were able to study the benefits of FiFo scheduling versus the original sweep schedule proposed in the CoEM. Figure 5.9c compares the number of updates required by both schedules to obtain a result of comparable quality on the larger dataset. Here we measure quality by L_1 parameter distance to an empirical estimate of the fixed point x^*, obtained by running a large number of iterations. For this application we do not find a substantial benefit from switching to a FiFo schedule.

We also investigated how GraphLab scales with problem size. Figure 5.9d shows the maximum speedup on 16 cpus attained with varying graph sizes, generated by sub-sampling a fraction of vertices from the large dataset. We find that parallel scaling improves with problem size and that even on smaller problems GraphLab is still able to achieve a factor of 12 speedup on 16 cores.

[1] Personal communication with Justin Betteridge and Tom Mitchell, Mar 12, 2010

Algorithm 5.5: CoEM Vertex Program for Named Entity Recognition

```
Input: Vertex data $\pi_v \in \mathcal{S}_v$ the distribution for this vertex.
Input: Vertex data $\{\pi_u : u \in \mathcal{N}_v\} \in \mathcal{S}_v$ the distributions for neighbors.
Input: Edge data $\{w_{u,v} : u \in \mathcal{N}_v\} \in \mathcal{S}_v$ weights based on co-occurrence
// Update the distribution on this vertex
$\pi_v^{\text{old}} \leftarrow \pi_v$
$\pi_v \leftarrow \sum_{u \in \mathcal{N}_v} w_{u,v} \pi_u$
// Determine which neighbors need to be recomputed
$\mathcal{T} \leftarrow \emptyset$
foreach $u \in \mathcal{N}_v$ do
    if $|\pi_v^{old} - \pi_v| w_{u,v} > \epsilon$ then
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{u\}$
return $\mathcal{T}$
```

Name	Classes	Verts.	Edges	Runtime
small	1	0.2 mil.	20 mil.	40 min
large	135	2 mil.	200 mil.	8 hours

(a) NER Datasets

Food	Religion	City
onion	Catholic	Munich
garlic	Fremasonry	Cape Twn.
noodles	Marxism	Seoul
blueberries	Catholic Chr.	Mexico Cty.
beans	Humanism	Winnipeg

(b) NER Types

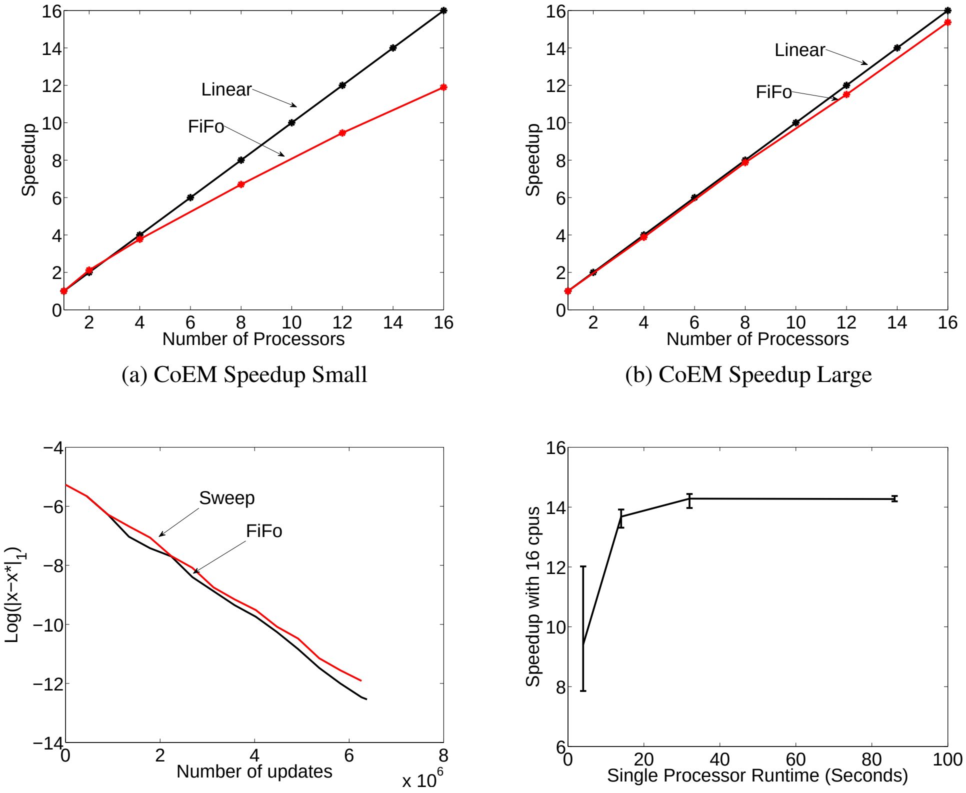
Table 5.2: **(a)** Dataset sizes for the CoEM benchmark and the corresponding single processor runtime. **(b)** Top words for several types.

5.6.4 Lasso

The Lasso [Tibshirani, 1996] is a popular feature selection and shrinkage method for linear regression which minimizes the objective $L(w) = \sum_{j=1}^{n}(w^T x_j - y_j)^2 + \lambda \left|\left|w\right|\right|_1$. At the time of this work there did not exist, a parallel algorithm for fitting a Lasso model. In this section we easily implement two different parallel algorithms for solving the Lasso. We use GraphLab to implement the Shooting Algorithm [Fu, 1998], a popular Lasso solver, and demonstrate that GraphLab is able to *automatically* obtain parallelism by identifying operations that can execute concurrently while retaining serializability.

The shooting algorithm works by iteratively minimizing the objective with respect to each dimension in w, corresponding to coordinate descent. We can formulate the Shooting algorithm in the GraphLab framework as a bipartite graph with a vertex for each weight w_i and a vertex for each observation y_j. An edge is created between w_i and y_j with weight $X_{i,j}$ if and only if $X_{i,j}$ is non-zero. We also define a vertex program (Alg. 5.6) which operates only on the weight vertices, and corresponds exactly to a single minimization step in the shooting algorithm. A sweep scheduling of Alg. 5.6 on all weight vertices corresponds exactly to the sequential shooting algorithm. We automatically obtain an equivalent parallel algorithm by select the full serializability model. Hence, by encoding the shooting algorithm in GraphLab we are able to discover a serializable *parallelization* based on the sparsity structure in our data.

We evaluate the performance of the GraphLab implementation on a financial data set obtained from Kogan



(a) CoEM Speedup Small

(b) CoEM Speedup Large

(c) Convergence

(d) Speedup with Problem Size

Figure 5.9: *CoEM Results* **(a,b)** Speedup measured relative to fastest running time on a single cpu for both datasets. The large dataset achieves better scaling because the increased number of types increases the computation time of each vertex program and hides most of the GraphLab overhead. **(c)** Speed of convergence measured in number of updates for FiFo and Sweep, **(d)** Speedup achieved with 16 cpus as the graph size is varied.

et al. [2009]. The task is to use word counts of a financial report to predict stock volatility of the issuing company for the consequent 12 months. Data set consists of word counts for 30K reports with the related stock volatility metrics.

To evaluate the affect of the various exclusion policies on the parallel scaling and algorithm convergence we create two datasets by deleting common words. The sparser dataset contains 209K features and 1.2M non-zero entries, and the denser dataset contains 217K features and 3.5M non-zero entries. The speedup curves are plotted in Figure 5.10. We observed better scaling (4x at 16 CPUs) on the sparser dataset than on the denser dataset (2x at 16 CPUs). This demonstrates that ensuring full serializability on denser graphs inevitably increases contention resulting in reduced scaling.

Additionally, we experimented with relaxing the serializability constraint by switching to the vertex exclusion policy. We discovered that the shooting algorithm still converges when serializability is no longer enforced; obtaining solutions with only 0.5% higher loss on the same termination criterion. The vertex exclusion model exposes substantially more parallelism and we therefore observe significantly better scaling, especially on the denser dataset.

Algorithm 5.6: Shooting Algorithm

```
Minimize the loss function with respect to $w_i$
if $w_i$ changed by $> \epsilon$ then
    Revise the residuals on all $y's$ connected to $w_i$
    Schedule all $w's$ connected to neighboring $y's$
end
```

Algorithm 5.7: Compressed Sensing Outer Loop

```
while $duality\_gap \geq \epsilon$ do
    Update edge and node data of the data graph.
    Use GraphLab to run GaBP on the graph
    Use aggregator to compute duality gap
    Take a newton step
end
```

5.6.5 Compressed Sensing

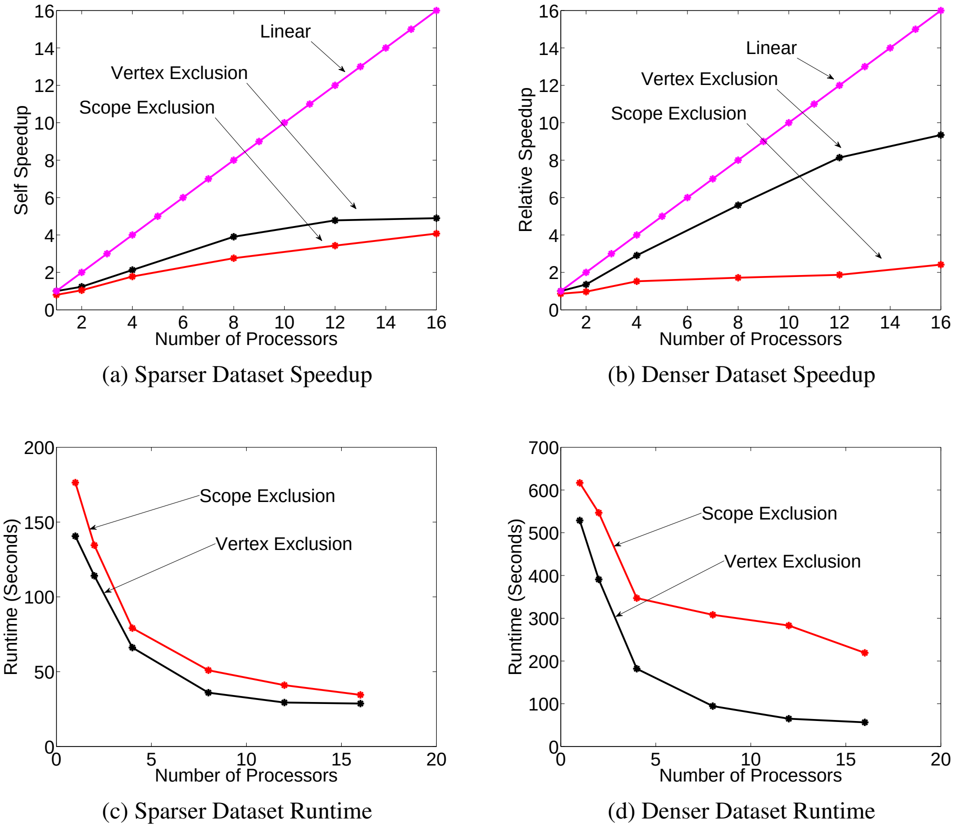
To show how GraphLab can be used as a subcomponent of a larger *sequential* algorithm, we implement a variation of the interior point algorithm proposed by Kim et al. [2007] for the purposes of compressed sensing. The aim is to use a sparse linear combination of basis functions to represent an image, while minimizing the reconstruction error. Sparsity is achieved through the use of elastic net regularization.

The interior point method is a double loop algorithm where the sequential outer loop (Alg. 5.7) implements a Newton method while the inner loop computes the Newton step by solving a sparse linear system using GraphLab. We used Gaussian BP (GaBP) as a linear solver [Bickson, 2008] since it has a natural GraphLab representation. The GaBP GraphLab construction follows closely the BP example in Section 5.6.1, but represents potentials and messages analytically as Gaussian distributions. In addition, the outer loop uses a Sync operation on the data graph to compute the duality gap and to terminate the algorithm when the gap falls below a predefined threshold. Because the graph structure is fixed across iterations, we can leverage data persistency in GraphLab, avoid both costly set up and tear down operations and resume from the converged state of the previous iteration.

We evaluate the performance of this algorithm on a synthetic compressed sensing dataset constructed by applying a random projection matrix to a wavelet transform of a 256 × 256 Lenna image (Figure 5.11). Experimentally, we achieved a factor of 8 speedup using 16 processors using sweep scheduling.

5.7 Distributed GraphLab Design

In this section we extend the shared memory design of the GraphLab system to the substantially more challenging distributed setting and discuss the techniques required to achieve this goal. An overview of the distributed design is illustrated in Figure 5.15a. Because of the inherently random memory access patterns common to dynamic asynchronous graph algorithms, we focus on the distributed **in-memory** setting, requiring the entire graph and all program state to reside in RAM.



(a) Sparser Dataset Speedup

(b) Denser Dataset Speedup

(c) Sparser Dataset Runtime

(d) Denser Dataset Runtime

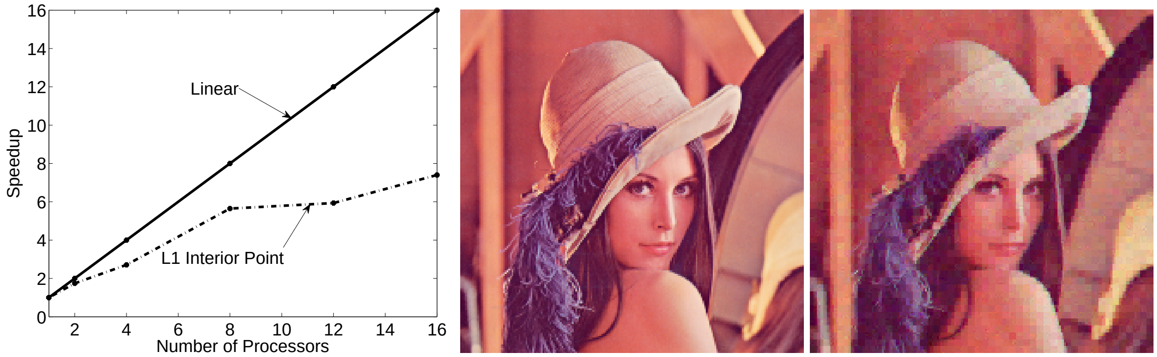
Figure 5.10: *Shooting Algorithm* **(a)** Speedup on the sparser dataset using *vertex exclusion* and *scope exclusion* (i.e., serializability) relative to the fastest single processor runtime. **(b)** Speedup on the denser dataset using *vertex exclusion* and *scope exclusion* relative to the fastest single processor runtime.

We first present our distributed data graph representation that allows for rapid repartitioning across different loads cluster sizes. Next, we describe two versions of the GraphLab engine for the distributed setting. The first engine is the simpler chromatic engine (Section 5.7.2) that uses basic graph coloring to achieve the various exclusion policies. The second is a locking engine (Section 5.7.2) that directly extends the lock based implementation of the exclusion policies to the distributed setting.

5.7.1 The Distributed Data Graph

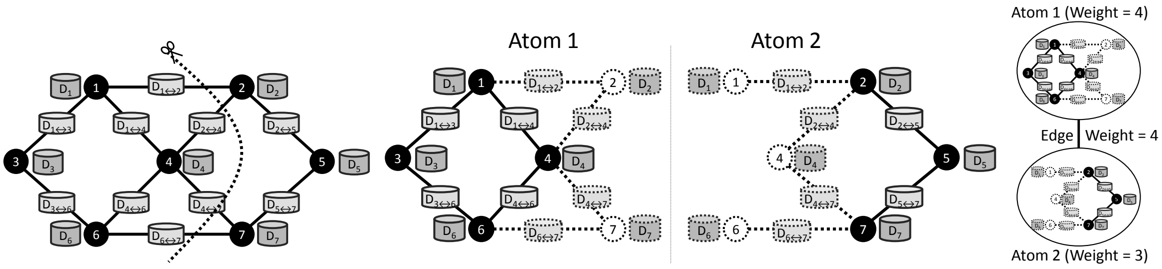
Efficiently implementing the data graph in the distributed setting requires balancing computation, communication, and storage. Therefore, we need to construct balanced partitioning of the data graph that minimize number of edges that cross between machines. Because the Cloud setting enables the size of the cluster to vary with budget and performance demands, we must be able to quickly load the data-graph on varying sized Cloud deployments. To resolve these challenges, we developed a graph representation based on two-phased partitioning which can be efficiently load balanced on a range of cluster sizes.

Motivated by our work on over partitioning for distributed belief propagation, the data graph is initially over-partitioned using domain specific knowledge (e.g., planar embedding), or by using a distributed graph



(a) Speedup (b) Lenna (c) Lenna 50%

Figure 5.11: **(a)** Speedup of the Interior Point algorithm on the compressed sensing dataset, **(b)** Original 256x256 test image with 65,536 pixels, **(c)** Output of compressed sensing algorithm using 32,768 random projections.



(a) Original Graph Cut (b) Resulting Atoms with Ghosts (c) Meta-Graph

Figure 5.12: *Partitions and Ghosts.* To represent a distributed graph we partition the graph into files each containing fragments of the original graph. In **(a)** we cut a graph into two (unbalanced) pieces. In **(b)** we illustrate the ghosting process for the resulting two atoms. The edges, vertices, and data illustrated with broken edges are **ghosts** in the respective atoms. The meta-graph **(c)** is a weighted graph of two vertices.

partitioning heuristic (e.g., ParMetis [Karypis and Kumar, 1998], Random Hashing) into k parts where k is much greater than the number of machines. Each part, called an **atom**, is stored as a separate file on a distributed storage system (e.g., HDFS, Amazon S3). Each atom file contains a subset of edges and vertices as well as the corresponding metadata. In addition, each atom stores information regarding **ghosts**: the set of vertices and edges adjacent to the partition boundary. The connectivity structure and file locations of the k atoms is stored in a **atom index** file as a **meta-graph** with k vertices (corresponding to the atoms) and edges encoding the connectivity of the atoms.

Distributed loading is accomplished by performing a fast balanced partition of the meta-graph over the number of physical machines. Each machine then constructs its local portion of the graph by playing back the journal from each of its assigned atoms. The playback procedure also instantiates the ghost of the local partition in memory. The ghosts are used as caches for their true counterparts across the network. Cache coherence is managed using a simple versioning system, eliminating the transmission of unchanged or constant data (e.g., edge weights).

The two-stage partitioning technique allows the same graph partition computation to be reused for different numbers of machines without requiring a full repartitioning step. Furthermore, random or even adaptive placement of the atoms can be used in situations where load balancing is needed (see Section 3.8.1). We evaluated two stage partitioning on a wide range of graphs obtained from Leskovec [2011] and found that if the initial atom graph is well partitioned then the two phase partitioning performs comparably. This is to be expected since typical graph partitioning algorithms perform a similar coarsening procedure.

The use of ghost vertices/edges in this design also introduces a limitation: the total amount of memory required by a single machine is not simply $\frac{1}{p}$ of the graph, but is a function of the graph structure. Dense graphs (e.g., a clique) could require every machine to store a copy of the complete graph. However, such dense graphs would violate the GrAD methodology and are not common in large-scale problems.

In our later experiments we found that certain graphs were too difficult for Metis/ParMetis to cut in reasonable time and memory. In these cases we resorted to random partitioning to provide the initial atoms. In the next chapter we will return to this problem and show that many real-world graphs, although sparse, have a particular sparsity pattern that will ultimately lead to costly replication and frustrate any edge cut based partitioning technique.

5.7.2 Distributed GraphLab Engines

The Distributed GraphLab **engine** emulates the *execution model* defined in Section 5.4.3 and is responsible for executing vertex programs and sync operations, maintaining the set of scheduled vertices $\mathcal{T}$, and ensuring serializability with respect to the appropriate exclusion policy (see Section 5.4.5). As discussed earlier, the precise order in which vertices are removed from $\mathcal{T}$ is up to the execution engine and scheduler and can affect performance and expressiveness. To evaluate this trade-off we built the low-overhead **Chromatic Engine**, which executes $\mathcal{T}$ partially asynchronously according to a fixed *sweep* schedule, and the more expressive **Locking Engine** which is fully asynchronous and supports relaxed versions of the *FiFo* and *priority* schedulers.

Chromatic Engine

In our work on parallel Gibbs sampling we introduced the Chromatic Gibbs sampler (Section 4.3) which uses graph color to achieve a serializable execution without the use of fine grained locking. The Chromatic GraphLab engine extends this classic technique to achieve a serializable parallel execution of vertex programs. Given a vertex coloring of the data graph, we can satisfy the *edge exclusion policy* by executing, *synchronously*, all vertices of the same color in the vertex set $\mathcal{T}$ before proceeding to the next color. We use the term **color-step**, in analogy to the *super-step* in the BSP model, to describe the process of updating all the vertices within a single color and communicating all changes. The sync operation can then be run safely between color-steps.

We can satisfy each of the exclusion policies simply by changing how the vertices are colored. The *scope exclusion policy* is satisfied by constructing a second-order vertex coloring (i.e., no vertex shares the same color as any of its distance two neighbors). The *vertex exclusion policy* is satisfied by assigning all vertices the same color. While optimal graph coloring is NP-hard in general, a reasonable quality coloring can be constructed quickly using graph coloring heuristics (e.g., greedy coloring). Furthermore, many MLDM problems produce graphs with trivial colorings. For example, many optimization problems in MLDM are naturally expressed as bipartite (two-colorable) graphs, while problems based upon template models can be easily colored using the template [Gonzalez et al., 2011].

Algorithm 5.8: Pipelined Locking Engine Thread Loop

```
while not done do
    while Lock acquisition pipeline is not full and the scheduler is not empty do
        Acquire a vertex $v$ from scheduler
        Add task to lock acquisition pipeline
    if Lock acquisition pipeline has ready vertex $v$ then
        Execute $(\mathcal{T}', \mathcal{S}_v) = f(v, \mathcal{S}_V)$
        // update scheduler on each machine
        For each machine $p$, Send $\{s \in \mathcal{T}' : \text{owner}(s) = p\}$
        Release locks and push changes to $\mathcal{S}_v$ in background
    else
        Wait on the lock acquisition pipeline
```

While the chromatic engine operates in synchronous color-steps, changes to ghost vertices and edges are communicated asynchronously as they are made. Consequently, the chromatic engine efficiently uses both network bandwidth and processor time within each color-step. However, we must ensure that all modifications are communicated before moving to the next color and therefore we require a synchronization barrier between color-steps.

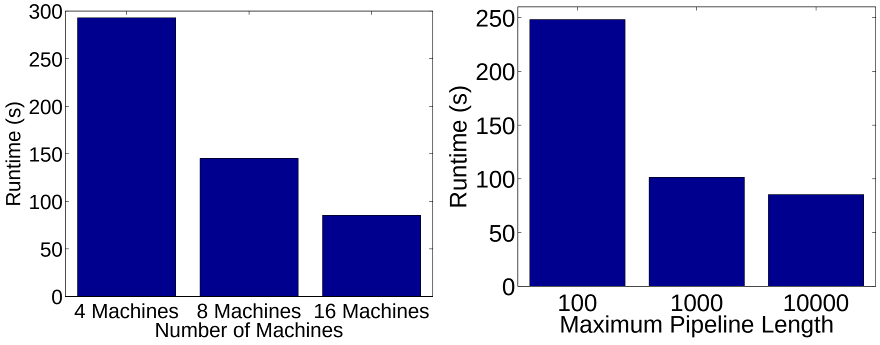
The simple design of the Chromatic engine is made possible by the explicit communication structure defined by the data graph, allowing data to be pushed directly to the machines requiring the information. In addition, the cache versioning mechanism further optimizes communication by only transmitting modified data. The advantage of the Chromatic engine lies its predictable execution schedule. Repeated invocations of the chromatic engine will always produce identical execution sequences, regardless of the number of machines used. This property makes the Chromatic engine highly suitable for testing and debugging purposes. The global barriers at the end of each color provide a natural breakpoint for debugging capabilities. We provide a distributed debugging tool that halts at the end of each color, allowing graph data and scheduler state to be queried.

The Chromatic engine supports limited dynamic scheduling. On each iteration, only active vertices are executed within each-color step and vertices can only activate vertices that are a different color. Moreover, the chromatic engine exactly preserves the semantics of the sweep schedule where vertices are ordered by color. As a consequence while the execution may be composed of synchronous sub-steps it actually corresponds to a fixed serial execution on the graph.

Distributed Locking Engine

While the Chromatic engine satisfies the GraphLab execution model and is capable of expressing dynamic computation, it does not provide sufficient scheduling flexibility (e.g., FiFo and Priority) for algorithms like SplashBP and Splash Gibbs sampling. To overcome these limitations, we introduce the distributed locking engine which extends the mutual exclusion technique used in the shared memory engine to the distributed setting.

Since the graph is partitioned, each machine is only responsible for scheduling and executing vertices that have been assigned to that machine. This enables each machine to run one of the many shared-memory



(a) Runtime (b) Pipeline Length

Figure 5.13: **(a)** Plots the runtime of the Distributed Locking Engine on a synthetic loopy belief propagation problem varying the number of machines with pipeline length $= 10,000$. **(b)** Plots the runtime of the Distributed Locking Engine on the same synthetic problem on 16 machines (128 CPUs), varying the pipeline length. Increasing pipeline length improves performance with diminishing returns.

schedulers allowing a much greater range of dynamic schedules. The ghost vertices and edges ensure that the update have direct memory access to all information in the scope. The ghosting system also eliminates the need to transmit or wait on data that has not changed remotely reducing network traffic and latency. Each worker thread on each machine evaluates the loop described in Alg. 5.8 until the scheduler is empty.

To hide the latency of distributed locking we pipeline the locking processess. When a vertex is removed from the scheduler its locks are requested and it is placed on the pipeline freeing the processor to execute other vertex programs or assist in lock processing. Once all the locks for a vertex have been acquired from remote machines the vertex is removed from the pipeline and executed. As locks are acquired the local ghost information is updated.

To evaluate the performance of the distributed pipelining system, we constructed a three-dimensional mesh of $300 \times 300 \times 300 = 27,000,000$ vertices. Each vertex is 26-connected (to immediately adjacent vertices along the axis directions, as well as all diagonals), producing over 375 million edges. The graph is partitioned using Metis [Karypis and Kumar, 1998] into 512 atoms. We interpret the graph as a binary Markov Random Field and evaluate the runtime of loopy belief propagation varying the length of the pipeline from 100 to 10,000, and the number of EC2 cluster compute instance (`cc1.4xlarge`) from 4 machines (32 processors) to 16 machines (128 processors). We observe in Figure 5.13a that the distributed locking system provides strong, nearly linear, scalability. In Figure 5.13b we evaluate the efficacy of the pipelining system by increasing the pipeline length. We find that increasing the length from 100 to 1000 leads to a *factor of three* reduction in runtime.

5.7.3 Fault Tolerance

We achieve fault tolerance in the distributed GraphLab framework using a distributed checkpoint mechanism. In the event of a failure, the system is recovered from the last checkpoint. We evaluate two strategies to construct distributed snapshots: a synchronous method that suspends all computation while the snapshot is constructed, and an asynchronous method that incrementally constructs a snapshot without suspending execution.

Algorithm 5.9: Snapshot Vertex Program on vertex v

```
if $v$ was already snapshotted then
    Quit
Save $D_v$ // Save current vertex
foreach $u \in \mathcal{N}_v$ do                                   // Loop over neighbors
    if $u$ was not snapshotted then
        Save data on edge $D_{u \leftrightarrow v}$
        Schedule $u$ for a Snapshot Update
Mark $v$ as snapshotted
```

The synchronous snapshots are easily constructed by suspending execution of vertex programs, flushing all communication channels, and then saving all modified data since the last snapshot. If a machine fails we simply restart the system with the data graph at the last checkout-point. Interestingly, we implemented the synchronous snapshot as a GraphLab aggregate *serializability enabled*.

Unfortunately, synchronous snapshots expose the GraphLab engine to the same inefficiencies of synchronous computation that GraphLab is trying to address. Using the GraphLab abstraction we designed and implemented a variant of the Chandy-Lamport [Chandy and Lamport, 1985] snapshot specifically tailored to the GraphLab data-graph and execution model. The resulting algorithm (Alg. 5.9) can be implemented as a *vertex program* and guarantees a consistent snapshot. The proof of correctness follows naturally from the original proof in Chandy and Lamport [1985] with the machines and channels replaced by vertices and edges and messages corresponding to scope modifications.

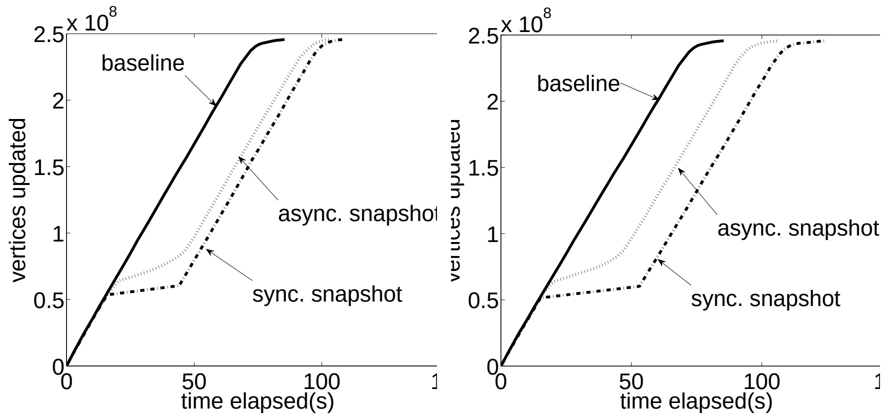
Both the synchronous and asynchronous snapshots are initiated at fixed intervals. The choice of interval must balance the cost of constructing the checkpoint with the computation lost since the last checkpoint in the event of a failure. Young et al. [Young, 1974] derived a first-order approximation to the optimal checkpoint interval:

$$T_{\text{Interval}} = \sqrt{2T_{\text{checkpoint}}T_{\text{MTBF}}} \tag{5.6}$$

where $T_{\text{checkpoint}}$ is the time it takes to complete the checkpoint and T_{MTBF} is the mean time between failures for the cluster. For instance, using a cluster of 64 machines, a per machine MTBF of 1 year, and a checkpoint time of 2 min leads to optimal checkpoint intervals of 3 hrs. Therefore, for the deployments considered in our experiments, even taking pessimistic assumptions for T_{MTBF}, leads to checkpoint intervals that far exceed the runtime of our experiments and in fact also exceed the Hadoop experiment runtimes. This brings into question the emphasis on strong fault tolerance in Hadoop. Better performance can be obtained by balancing fault tolerance costs against that of a job restart.

We evaluate the performance of the snapshotting algorithms on the same synthetic mesh problem described in the previous section, running on 16 machines (128 processors). We configure the implementation to issue exactly one snapshot in the middle of the second iteration. In Figure 5.14a we plot the number of updates completed against time elapsed. The effect of the synchronous snapshot and the asynchronous snapshot can be clearly observed: synchronous snapshots stops execution, while the asynchronous snapshot only slows down execution.

The benefits of asynchronous snapshots become more apparent in the **multi-tenancy** setting where variation in system performance exacerbate the cost of synchronous operations. We simulate this on Amazon EC2 by halting one of the processes for 15 seconds after snapshot begins. In figures Figure 5.14b we again plot the number of updates completed against time elapsed and we observe that the asynchronous snapshot is



(a) Snapshot

(b) Snapshot with Delay

Figure 5.14: **(a)** The number of vertices updated vs. time elapsed for 10 iterations comparing asynchronous and synchronous snapshots. Synchronous snapshots (completed in 109 seconds) have the characteristic "flatline" while asynchronous snapshots (completed in 104 seconds) allow computation to proceed. **(b)** Same setup as in (a) but with a single machine fault lasting 15 seconds. As a result of the 15 second delay the asynchronous snapshot incurs only a 3 second penalty while the synchronous snapshot incurs a 16 second penalty.

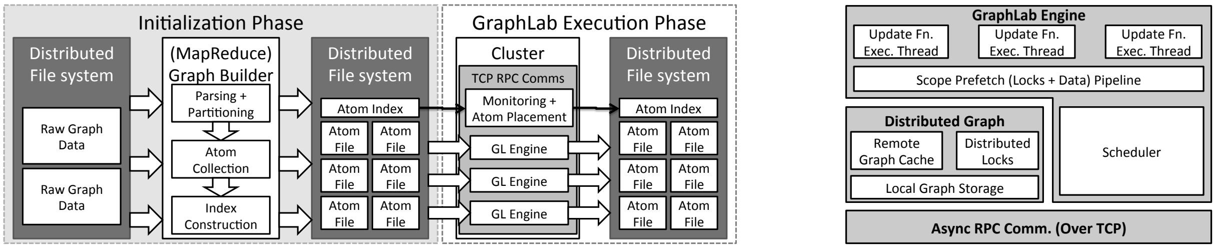
minimally affected by the simulated failure (adding only 3 seconds to the runtime), while the synchronous snapshot experiences a full 15 second increase in runtime.

5.7.4 System Design

In Figure 5.15, we provide a high-level overview of the design of the distributed GraphLab system. The user begins by constructing the atom graph representation on a Distributed File System (DFS). When GraphLab is launched on a cluster, one instance of the GraphLab program is executed on each machine. The GraphLab processes are symmetric and directly communicate with each other using a custom asynchronous RPC protocol over TCP/IP. The first process has the additional responsibility of being a master/monitoring machine.

At launch the master process computes the placement of the atoms based on the atom index, following which all processes perform a parallel load of the atoms they were assigned. Each process is responsible for a partition of the distributed graph that is managed within a local graph storage, and provides distributed locks. A cache is used to provide access to remote graph data.

Each process also contains a scheduler that manages the vertices in $\mathcal{T}$ that have been assigned to the process. At runtime, each machine's local scheduler feeds vertices into a prefetch pipeline that collects the data and locks required to execute the vertex. Once all data and locks have been acquired, the vertex is executed by a pool of worker threads. Vertex scheduling is decentralized with each machine managing the schedule for its local vertices and forwarding scheduling requests for remote vertices. Finally, a distributed consensus algorithm [Misra, 1983] is used to determine when all schedulers are empty. Due to the symmetric design of the distributed runtime, there is no centralized bottleneck.



(a) System Overview

(b) Locking Engine Design

Figure 5.15: **(a)** A high level overview of the GraphLab system. In the initialization phase the atom file representation of the data graph is constructed. In the GraphLab Execution phase the atom files are assigned to individual execution engines and are then loaded from the DFS. **(b)** A block diagram of the parts of the Distributed GraphLab process. Each block in the diagram makes use of the blocks below it. For more details, see Section 5.7.4.

5.8 Distributed GraphLab Evaluation

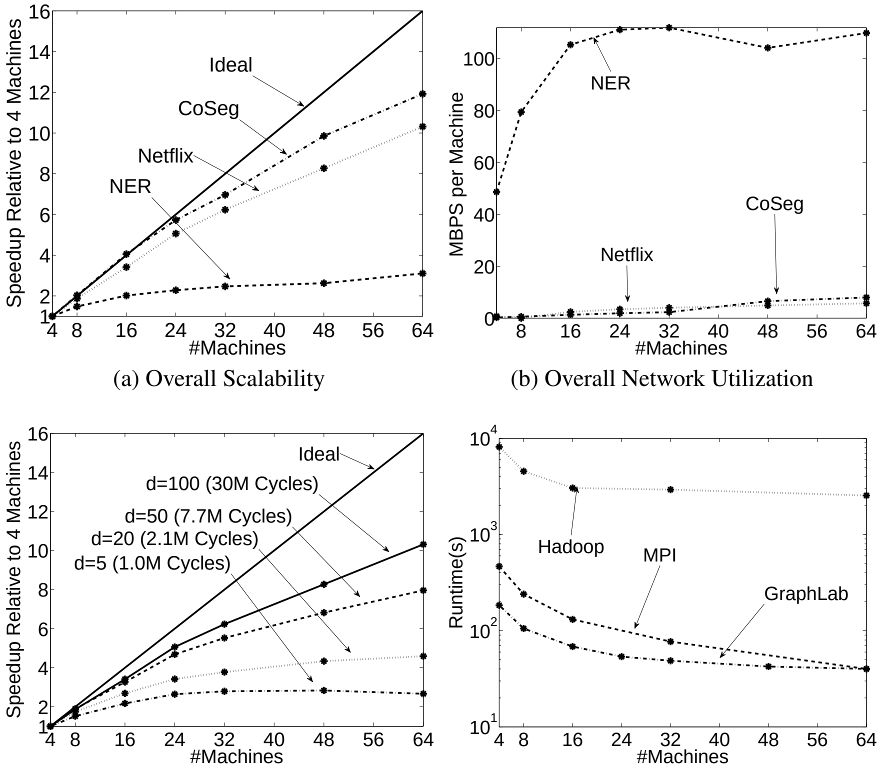
We evaluated the distributed implementation of GraphLab on three state-of-the-art MLDM applications: collaborative filtering for Netflix movie recommendations, Video Co-segmentation (CoSeg) and Named Entity Recognition (NER). Each experiment was based on large real-world problems and datasets (see Table 5.3). We used the Chromatic engine for the Netflix and NER applications and the Locking Engine for the CoSeg application. Equivalent Hadoop and MPI implementations were also evaluated on the Netflix and NER applications. All timing experiments include loading time and atom graph partitioning.

Experiments were performed on Amazon's Elastic Computing Cloud (EC2) using up to 64 High-Performance Cluster (HPC) instances (`cc1.4xlarge`) each with dual Intel Xeon X5570 quad-core Nehalem processors and 22 GB of memory and connected by a 10 Gigabit Ethernet network. All timings include data loading and are averaged over three or more runs. On each node, GraphLab spawns eight engine threads (matching the number of cores).

In Figure 5.16a we present an aggregate summary of the parallel speedup of GraphLab when run on 4 to 64 HPC machines on all three applications. In all cases, speedup is measured relative to the *four node deployment* since single node experiments were not always feasible due to memory limitations. No snapshots were constructed during the timing experiments since all experiments completed prior to the first snapshot under the optimal snapshot interval (3 hours) as computed in Section 5.7.3. To provide intuition regarding the snapshot cost, in Figure 5.18d we plot for each application, the overhead of compiling a snapshot on a 64 machine cluster.

Our principle findings are:

- On equivalent tasks, GraphLab outperforms Hadoop by 20-60x and performance is comparable to tailored MPI implementations.
- GraphLab's performance scaling improves with higher computation to communication ratios.
- The GraphLab abstraction more compactly expresses the Netflix, NER and Coseg algorithms than MapReduce or MPI.



(a) Overall Scalability

(b) Overall Network Utilization

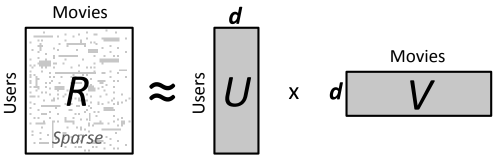
(c) Netflix Scaling with Intensity

(d) Netflix Comparisons

Figure 5.16: **(a)** Scalability of the three test applications with the largest input size. CoSeg scales excellently due to very sparse graph and high computational intensity. Netflix with default input size scales moderately while NER is hindered by high network utilization. **(b)** Average bandwidth utilization per cluster node. Netflix and CoSeg have very low bandwidth requirements while NER appears to saturate when #machines > 24. **(c)** Scalability of Netflix varying the computation cost of the vertex program. **(d)** Runtime of Netflix with GraphLab, Hadoop and MPI implementations. Note the logarithmic scale. GraphLab outperforms Hadoop by 40-60x and is comparable to an MPI implementation. See Section 5.8.1 and Section 5.8.3 for a detailed discussion.

5.8.1 Netflix Movie Recommendation

The Netflix movie recommendation task uses *collaborative filtering* to predict the movie ratings for each user, based on the ratings of similar users. We implemented the **alternating least squares** (**ALS**) algorithm Zhou et al. [2008], a common algorithm in collaborative filtering. The input to ALS is a sparse users by movies matrix R, containing the movie ratings of each user. The algorithm iteratively computes a low-rank matrix factorization:



(5.7)

where U and V are rank d matrices. The ALS algorithm alternates between computing the least-squares solution for U and V while holding the other fixed. Both the quality of the approximation and the computational complexity depend on the magnitude of d: higher d produces higher accuracy while increasing computational cost. Collaborative filtering and the ALS algorithm are important tools in MLDM: an effective solution for ALS can be extended to a broad class of other applications.

While ALS may not seem like a graph algorithm, it can be represented easily using the GraphLab abstraction. The *sparse* matrix R defines a bipartite graph connecting each user with the movies they rated. The edge data contains the rating for a movie-user pair. The vertex data for users and movies contains the corresponding row in U and column in V respectively. The GraphLab vertex program recomputes the d length vector for each vertex by reading the d length vectors on adjacent vertices and then solving a least-squares regression problem to predict the edge values. The chromatic engine is used since the graph is bipartite, two colorable, and the edge serializability model is sufficient for serializability.

The Netflix task provides us with an opportunity to quantify the distributed chromatic engine overhead since we are able to directly control the computation-communication ratio by manipulating d: the dimensionality of the approximating matrices in Eq. (5.7). In Figure 5.16c we plot the speedup achieved for varying values of d and the corresponding number of cycles required per update. Extrapolating to obtain the theoretically optimal runtime, we estimated the overhead of Distributed GraphLab at 64 machines (512 CPUs) to be about 12x for $d = 5$ and about 4.9x for $d = 100$. Note that this overhead includes graph loading and communication. This provides us with a measurable objective for future optimizations.

Next, we compare against a Hadoop and an MPI implementation in Figure 5.16d ($d = 20$ for all cases), using between 4 to 64 machines. The Hadoop implementation is part of the Mahout [2] project and is widely used. Since fault tolerance was not needed during our experiments, we reduced the Hadoop Distributed Filesystem's (HDFS) replication factor to one. A significant amount of our effort was then spent tuning the Hadoop job parameters to improve performance. However, even so we find that GraphLab performs between **40-60** times faster than Hadoop.

While some of the Hadoop inefficiency may be attributed to Java, job scheduling, and various design decisions, GraphLab also leads to a more efficient representation of the underlying algorithm. We can observe that the Map function of a Hadoop ALS implementation, performs no computation and its only purpose is to emit *copies* of the vertex data for every edge in the graph; unnecessarily multiplying the amount of data that need to be tracked. For example, a user vertex that connects to 100 movies must emit the data on the user vertex 100 times, once for each movie. This results in the generation of a large amount of unnecessary network traffic and unnecessary HDFS writes.

This weakness extends beyond the MapReduce abstraction and affects the graph message-passing models (such as Pregel) due to the lack of a *scatter* operation that would avoid sending same value multiple times to each machine. Comparatively, the GraphLab vertex program is simpler as users do not need to explicitly define the flow of information. Synchronization of a modified vertex only requires as much communication as there are ghosts of the vertex. In particular, only machines that require the vertex data for computation will receive it, and each machine receives each modified vertex data at most once, even if the vertex has many neighbors.

Our MPI implementation of ALS is relatively optimized, and uses synchronous MPI collective operations for communication. The computation is broken into super-steps that alternate between recomputing the latent user and movies low rank matrices. Between super-steps the new user and movie values are scattered (using MPI_Alltoall) to the machines that need them in the next super-step. As a consequence our MPI implementation of ALS is roughly equivalent to an optimized Pregel version of ALS with added support for

[2] https://issues.apache.org/jira/browse/MAHOUT-542

parallel broadcasts. Surprisingly, GraphLab was able to outperform the MPI implementation. We attribute the performance to the use of background asynchronous communication in GraphLab.

Finally, we can evaluate the effect of enabling dynamic computation. In Figure 5.19a, we plot the test error obtained over time using a dynamic update schedule as compared to a static BSP-style update schedule. This dynamic schedule is easily represented in GraphLab while it is difficult to express using Pregel messaging semantics. We observe that a dynamic schedule converges much faster, reaching a low test error using less than half the amount of work.

5.8.2 Video Co-segmentation (CoSeg)

Video co-segmentation automatically identifies and clusters spatio-temporal segments of video (Figure 5.17a) that share similar texture and color characteristics. The resulting segmentation (Figure 5.17a) can be used in scene understanding and other computer vision and robotics applications. Previous co-segmentation methods Batra et al. [2010] have focused on processing frames in isolation. Instead, we developed a joint co-segmentation algorithm that processes all frames concurrently and is able to model temporal stability.

We preprocessed 1,740 frames of high-resolution video by coarsening each frame to a regular grid of 120×50 rectangular **super-pixels**. Each super-pixel stores the color and texture statistics for all the raw pixels in its domain. The CoSeg algorithm predicts the best label (e.g., sky, building, grass, pavement, trees) for each super pixel using **Gaussian Mixture Model** (**GMM**) in conjunction with belief propagation (Chapter 3). The GMM estimates the best label given the color and texture statistics in the super-pixel. The algorithm operates by connecting neighboring pixels in time and space into a large three-dimensional grid and uses belief propagation to smooth the local estimates. We combined the two algorithms to form an *asynchronous* Expectation-Maximization algorithm, which concurrently estimates the label for each super-pixel given the GMM and updates the GMM given the labels from belief propagation.

The GraphLab vertex program executes the belief propagation iterative updates. We implement the state-of-the-art adaptive belief residual schedule described in Chapter 3, where updates that are expected to change vertex values significantly are prioritized. We therefore make use of the locking engine with a priority scheduler. The parameters for the GMM are maintained using the sync operation. To the best of our knowledge, there are no other abstractions that provide the dynamic asynchronous scheduling as well as the sync (reduction) capabilities required by this application.

In Figure 5.16a we demonstrate that the locking engine can achieve scalability and performance on the large 10.5 million vertex graph used by this application, resulting in a 10x speedup with 16x more machines. We also observe from Figure 5.18a that the locking engine provides nearly optimal weak scaling: the runtime does not increase significantly as the size of the graph increases proportionately with the number of machines. We can attribute this to the properties of the graph partition where the number of edges crossing machines increases linearly with the number of machines, resulting in low communication volume.

While Section 5.7.2 contains a limited evaluation of the pipelining system on a synthetic graph, here we further investigate the behavior of the distributed lock implementation when run on a complete problem that makes use of all key aspects of GraphLab: both sync and dynamic prioritized scheduling. The evaluation is performed on a small 32-frame (192K vertices) problem using a 4 node cluster and two different partitioning. An *optimal partition* was constructed by evenly distributing 8 frame blocks to each machine. A *worst case partition* was constructed by striping frames across machines and consequently stressing the distributed lock implementation by forcing each scope acquisition is to grab at least one remote lock. We also vary the maximum length of the pipeline. Results are plotted in Figure 5.18b. We demonstrate that increasing the length of the pipeline increases performance significantly and is able to compensate for poor partitioning, rapidly bringing down the runtime of the problem. Just as in Section 5.7.2, we observe diminishing returns with increasing pipeline length.

We conclude that for the video co-segmentation task, Distributed GraphLab provides excellent performance while being the only distributed graph abstraction that allows the use of dynamic prioritized scheduling. In addition, the pipelining system is an effective way to hide latency, and to some extent, a poor partitioning.

5.8.3 Named Entity Recognition (NER)

Here we return to the NER task described in the multicore experiments in Section 5.6.3 using the same data and vertex program. Since the graph is two colorable and relatively dense the chromatic engine was used with random partitioning. The lightweight floating point arithmetic in the NER computation in conjunction with the relatively dense graph structure and random partitioning is essentially the worst-case for the current Distributed GraphLab design, and thus allow us to evaluate the overhead of the Distributed GraphLab runtime.

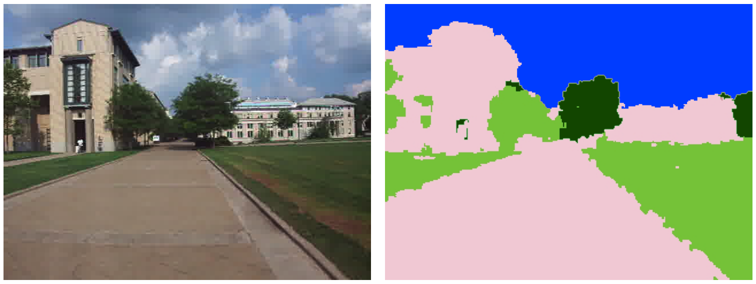
From Figure 5.16a we see that NER achieved only a modest 3x improvement using 16x more machines. We attribute the poor scaling performance of NER to the large vertex data size (816 bytes), dense connectivity, and poor partitioning (random cut) that resulted in substantial communication overhead per iteration. Figure 5.16b shows for each application, the average number of bytes per second transmitted by each machine with varying size deployments. On four nodes, NER transmits over 10 times more data than Netflix and almost 100 times as much data as CoSeg. Beyond 16 machines, NER saturates with each machine sending at a rate of over 100MB per second.

We evaluated our Distributed GraphLab implementation against a Hadoop and an MPI implementation in Figure 5.18c. In addition to the optimizations listed in Section 5.8.1, our Hadoop implementation required the use of binary serialization methods to obtain reasonable performance (decreasing runtime by 5x from baseline). We demonstrate that GraphLab implementation of NER was able to obtains a 20-30x speedup over Hadoop. The reason for the performance gap is the same as that for the Netflix evaluation. Since each vertex emits a copy of itself for each edge: in the extremely large CoEM graph, this corresponds to over 100 GB of HDFS writes occurring between the Map and Reduce stage.

On the other hand, our MPI implementation was able to outperform Distributed GraphLab. The CoEM task requires extremely little computation in comparison to the amount of data it touches. We were able to evaluate that the NER vertex program requires 5.7x fewer cycles per byte of data accessed as compared to the Netflix problem at $d = 5$ (the hardest Netflix case evaluated). The extremely poor computation to communication ratio stresses our communication implementation, that is outperformed by MPI's efficient communication layer. Furthermore, Figure 5.16b provides further evidence that we fail to fully saturate the network (that offers 10Gbps). Further optimizations to eliminate inefficiencies in GraphLab's communication layer should bring us up to parity with the MPI implementation.

Exp.	#Verts	#Edges	Vert. Data	Edge Data	Complexity	Shape	Partition	Engine
Netflix	0.5M	99M	$8d + 13$	16	$O(d^3 + deg.)$	bipartite	random	Chromatic
CoSeg	10.5M	31M	392	80	$O(deg.)$	3D grid	frames	Locking
NER	2M	200M	816	4	$O(deg.)$	bipartite	random	Chromatic

Table 5.3: *Experiment input sizes.* The vertex and edge data are measured in bytes and the d in Netflix is the size of the latent dimension.


(a) Coseg Video Frame

(b) Segmentation

Figure 5.17: *Coseg:* **(a)** A frame from the original video sequence and the result of running the co-segmentation algorithm. **(b)** The predicted segmentation.

5.8.4 EC2 Cost evaluation

To illustrate the monetary cost of using the alternative abstractions, we plot the price-runtime curve for the Netflix application in Figure 5.19b in *log-log* scale. All costs are computed using fine-grained billing rather than the hourly billing used by Amazon EC2. The price-runtime curve demonstrates diminishing returns: the cost of attaining reduced runtimes increases faster than linearly. As a comparison, we provide the price-runtime curve for Hadoop on the same application. For the Netflix application, GraphLab is about *two orders of magnitude* more cost-effective than Hadoop.

5.9 Additional Related Work

Section 5.3 provides a detailed comparison of several contemporary high-level parallel and distributed frameworks. In this section we review related work in classic parallel abstractions, graph databases, and domain specific languages.

There has been substantial work [Angles and Gutierrez, 2008] in graph structured databases dating back to the 1980's along with several recent open-source and commercial products (e.g., Neo4j [2011]). Graph databases typically focus on efficient storage and retrieval of graph structured data with support for basic graph computation. In contrast, GraphLab focuses GrAD style computation on large graphs.

There are several notable projects focused on using MapReduce for graph computation. Pegasus [Kang et al., 2009] is a collection of algorithms for mining large graphs using Hadoop. Surfer [Chen et al., 2010] extends MapReduce with a *propagation* primitive, but does not support asynchronous or dynamic scheduling. Alternatively, Lattanzi et al. [2011] proposed subsampling large-graphs (using MapReduce) to a size which can be processed on a single machine. While Lattanzi et al. [2011] was able to derive reductions for some graph problems (e.g., minimum spanning tree), the techniques are not easily generalizable and may not be applicable to many MLDM algorithms.

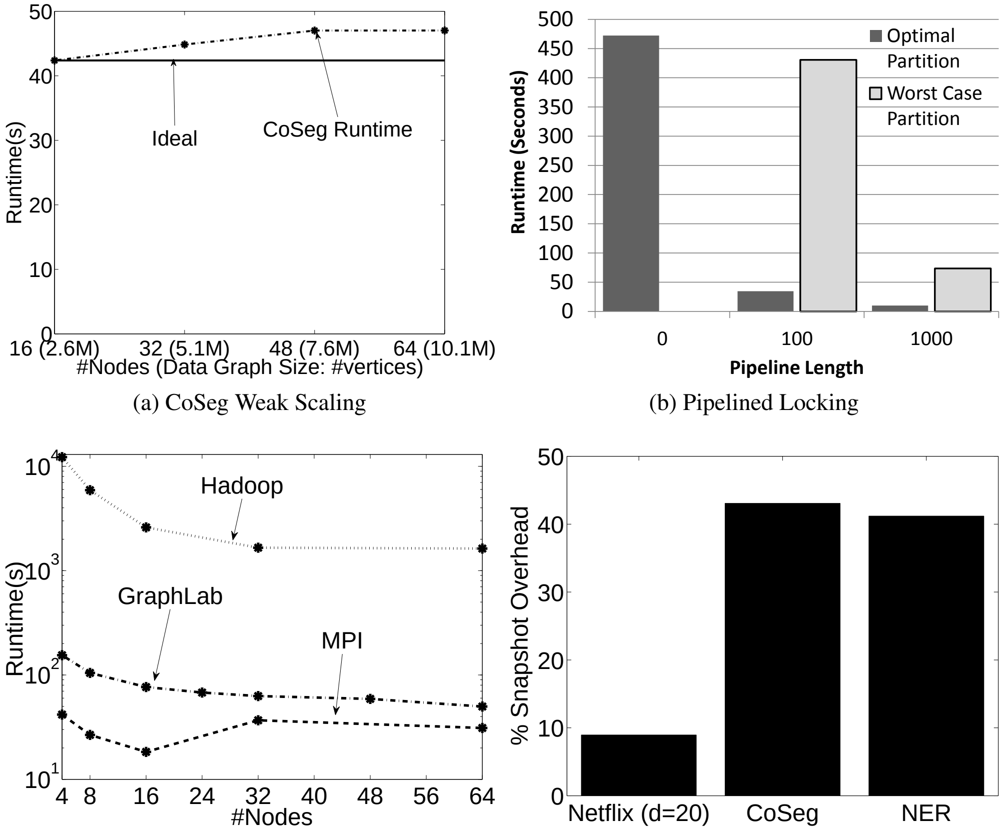
5.10 Conclusion

The GrAD methodology emphasizes the importance of explicitly modeling sparse computational dependencies, asynchronous computation, and dynamic scheduling in large scale MLDM problems. In this chapter we described how existing high-level parallel and distributed abstractions fail to support all three critical properties of the GrAD methodology. To address these properties we introduced the GraphLab abstraction, a graph-parallel framework that explicitly targets GrAD algorithms and generalizes our earlier work on parallel belief propagation and Gibbs sampling.

The GraphLab abstraction introduces the **data graph** to decompose the program state along the sparse graphical dependency structuring. User defined computation in GraphLab is expressed in the form of **vertex programs** that run *asynchronously* on each vertex and can directly read and modify values on neighboring edges and vertices. GraphLab supports a rich set of **schedulers** that allow vertex programs to *dynamically* trigger and *prioritize* computation on neighboring vertices. To simplify reasoning about highly parallel and distributed asynchronous executions, GraphLab introduces a set of simple mutual **exclusion policies** which ensure a serializable execution. As a consequence, we can often directly apply serial analysis techniques to programs implemented within the GraphLab abstraction. Finally, to manage sharing and aggregation of global state, GraphLab provides a powerful **aggregation framework**.

We developed optimized shared-memory and distributed implementations of the GraphLab abstraction by extending the techniques used to build scalable systems for belief propagation and Gibbs sampling. In the distributed setting we introduced a novel pipelined locking protocol and multistage partitioning and were able to achieve faul-tolerance by directly exploiting the key serializability properties of the abstraction. We evaluated our implementation on multi-core and distributed systems and demonstrated strong scaling and state-of-the-art performance on real-world applications. By implementing a wide range of algorithms we demonstrated the expressiveness of the GraphLab abstraction within the context of a single MLDM application. In the distributed setting we demonstrated that GraphLab significantly outperforms Hadoop by 20-60x, and is competitive with application specific MPI implementations.

The GraphLab API has the opportunity to be an interface between the ML and systems communities. Parallel ML algorithms built around the GraphLab API automatically benefit from developments in parallel data structures. As new locking protocols and parallel scheduling primitives are incorporated into the GraphLab API, they become immediately available to the ML community. Systems experts can more easily port ML algorithms to new parallel hardware by porting the GraphLab API. Future implementations of GraphLab will benefit from research in system architectures and parallel data structures. These gains are automatically conferred to all algorithms built in the GraphLab framework.



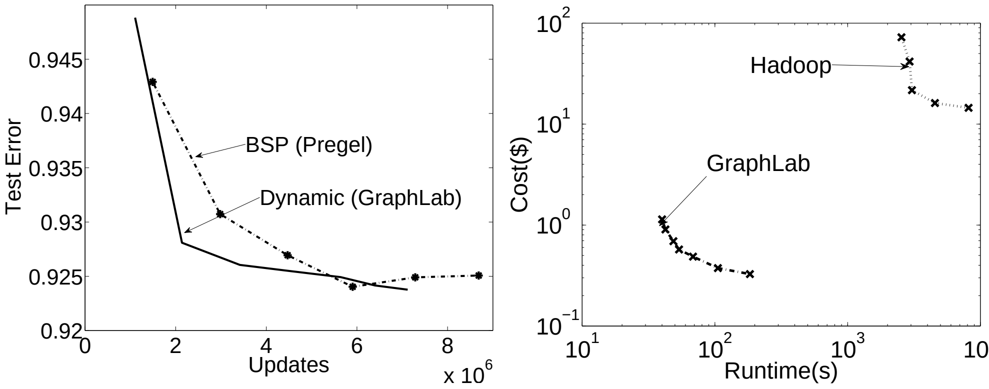
(a) CoSeg Weak Scaling

(b) Pipelined Locking

(c) NER Comparisons

(d) Snapshot Overhead

Figure 5.18: **(a)** Runtime of the CoSeg experiment as data set size is scaled proportionately with the number of machines. Ideally, runtime is constant. GraphLab experiences an 11% increase in runtime scaling from 16 to 64 machines. **(b)** The performance effects of varying the length of the pipeline. Increasing the pipeline length has a small effect on performance when partitioning is good. When partitioning is poor, increasing the pipeline length improves performance to be comparable to that of optimal partitioning. Runtime for worst-case partitioning at pipeline length 0 is omitted due to excessive runtimes. **(c)** Runtime of the NER experiment with Distributed GraphLab, Hadoop and MPI implementations. Note the logarithmic scale. GraphLab outperforms Hadoop by about 80x when the number of machines is small, and about 30x when the number of machines is large. The performance of Distributed GraphLab is comparable to the MPI implementation. **(d)** For each application, the overhead of performing a complete snapshot of the graph every $|V|$ updates (where $|V|$ is the number of vertices in the graph), when running on a 64 machine cluster.



(a) Dynamic Netflix

(b) EC2 Price/Performance

Figure 5.19: **(a)** Convergence rate when dynamic computation is used. Dynamic computation can converge to equivalent test error in about half the number of updates. **(b)** Price Performance ratio of GraphLab and Hadoop on running on Amazon EC2 HPC nodes in a log-log scale. Costs were computing assuming fine-grained billing.

Chapter 6

PowerGraph: Scaling GraphLab to the Web and Beyond

In the previous chapters we discussed how the GrAD methodology can lead to efficient large-scale probabilistic reasoning on graph-structured data. We then introduced the GraphLab abstraction which generalizes the GrAD methodology, simplifying the design and implementation of parallel and distributed graph-parallel computation. However, in the distributed setting GraphLab began to struggle on web-scale graphs and even on smaller graphs with light-weight computation (e.g., CoEM Section 5.8.3) we found that traditional graph partitioning techniques failed and communication overhead dominated. Upon deeper analysis we discovered that the root cause of these limitations arrises from the power-law sparsity structure common to a wide range or real-world graphs. Furthermore, these limitations are not unique to GraphLab but instead are present in a wide range of existing parallel abstractions.

In this chapter, we characterize the challenges of computation on natural graphs in the context of existing graph-parallel abstractions. We then introduce the PowerGraph abstraction which exploits the internal structure of graph programs to address these challenges. Leveraging the PowerGraph abstraction we introduce a new approach to distributed graph placement and representation that exploits the structure of power-law graphs. We provide a detailed analysis and experimental evaluation comparing PowerGraph to two popular graph-parallel systems. Finally, we describe three different implementation strategies for PowerGraph and discuss their relative merits with empirical evaluations on large-scale real-world problems demonstrating order of magnitude gains.

6.1 Introduction

The increasing need to reason about large-scale graph-structured data in machine learning and data mining (MLDM) presents a critical challenge. As the sizes of datasets grow, statistical theory suggests that we should apply richer models to eliminate the unwanted bias of simpler models, and extract stronger signals from data. At the same time, the computational and storage complexity of richer models coupled with rapidly growing datasets have exhausted the limits of single machine computation.

The resulting demand has driven the development of new *graph-parallel* abstractions such as Pregel [Malewicz et al., 2010] and GraphLab that encode computation as *vertex-programs* which run in parallel and interact along edges in the graph. Graph-parallel abstractions rely on each vertex having a small neighborhood to maximize parallelism and effective partitioning to minimize communication. However, graphs derived from real-world phenomena, like social networks and the web, typically have *power-law* degree distributions, which implies that a small subset of the vertices connects to a large fraction of the graph. Furthermore, power-law graphs are difficult to partition [Abou-Rjeili and Karypis, 2006, Leskovec et al., 2008] and represent in a distributed environment.

To address the challenges of power-law graph computation, we introduce the PowerGraph abstraction which exploits the structure of vertex-programs and explicitly factors computation over edges instead of vertices. As a consequence, PowerGraph exposes substantially greater parallelism, reduces network communication and storage costs, and provides a new highly effective approach to distributed graph placement. We describe the design of our distributed implementation of PowerGraph and evaluate it on a large EC2 deployment using real-world applications. In particular our key contributions are:

1. An analysis of the challenges of power-law graphs in distributed graph computation and the limitations of existing graph parallel abstractions (Sec. 6.2 and 6.3).
2. The PowerGraph abstraction (Section 6.4) which factors individual vertex-programs.
3. A delta caching procedure which allows computation state to be dynamically maintained (Section 6.4.2).
4. A new fast approach to data layout for power-law graphs in distributed environments (Section 6.5).
5. An theoretical characterization of network and storage (Theorem 6.5.2, Theorem 6.5.3).
6. A high-performance open-source implementation of the PowerGraph abstraction (Section 6.7).
7. A comprehensive evaluation of three implementations of PowerGraph on a large EC2 deployment using real-world MLDM applications (Section 6.6 and 6.7).

6.2 Graph-Parallel Abstractions

A **graph-parallel** abstraction consists of a *sparse* graph $G = \{V, E\}$ and a **vertex-program** Q which is executed in parallel on each vertex $v \in V$ and can interact (e.g., through shared-state in GraphLab, or messages in Pregel) with neighboring instances $Q(u)$ where $(u, v) \in E$. In contrast to more general message passing models, graph-parallel abstractions constrain the interaction of vertex-program to a graph structure enabling the optimization of data-layout and communication. We focus our discussion on Pregel and GraphLab as they are representative of existing graph-parallel abstractions.

6.2.1 Pregel

Pregel, introduced by Malewicz et al. [2010], is a bulk synchronous *message passing* abstraction in which all vertex-programs run simultaneously in a sequence of super-steps. Within a **super-step** each program instance $Q(v)$ receives all messages from the previous super-step and sends messages to its neighbors in the next super-step. A barrier is imposed between super-steps to ensure that all program instances finish processing messages from the previous super-step before proceeding to the next. The program terminates when there are no messages remaining and every program has voted to halt. Pregel introduces commutative associative message **combiners** which are user defined functions that merge messages destined to the same vertex. The following is an example of the PageRank vertex-program implemented in Pregel. The vertex-program receives the single incoming message (after the combiner) which contains the sum of the PageRanks of all in-neighbors. The new PageRank is then computed and sent to its out-neighbors.

```
Message combiner(Message m1, Message m2) :
  return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
  float total = msg.value();
  vertex.val = 0.15 + 0.85*total;
  foreach(nbr in out_neighbors) :
     SendMsg(nbr, vertex.val/num_out_nbrs);
```

6.2.2 GraphLab

GraphLab (Chapter 5) is an *asynchronous distributed shared-memory* abstraction in which vertex-programs have shared access to a distributed graph with data stored on every vertex and edge. Each vertex-program may directly access information on the current vertex, adjacent edges, and adjacent vertices irrespective of edge direction. Vertex-programs can schedule neighboring vertex-programs to be executed in the future. GraphLab ensures serializability by preventing neighboring program instances from running simultaneously. The following is an example of the PageRank vertex-program implemented in GraphLab. The GraphLab vertex-program directly reads neighboring vertex values to compute the sum.

```
void GraphLabPageRank(Scope scope) :
  float accum = 0;
  foreach (nbr in scope.in_nbrs) :
     accum += nbr.val / nbr.nout_nbrs();
  vertex.val = 0.15 + 0.85 * accum;
```

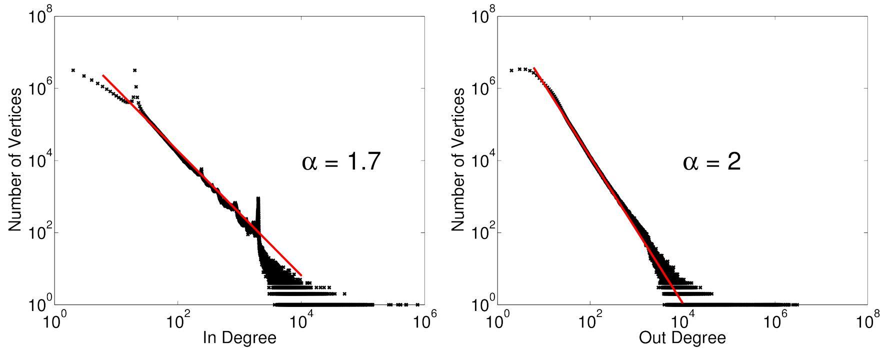
By eliminating messages, GraphLab isolates the user defined algorithm from the movement of data, allowing the system to choose when and how to move program state. By allowing mutable data to be associated with both vertices *and edges* GraphLab allows the algorithm designer to more precisely distinguish between data shared with all neighbors (vertex data) and data shared with a particular neighbor (edge data).

6.2.3 GAS Decomposition

While the implementation of MLDM vertex-programs in GraphLab and Pregel differ in how they collect and disseminate information, they share a common overall structure. To characterize this common structure and differentiate between vertex and edge specific computation we introduce the GAS model of graph computation.

The **GAS** model represents three *conceptual* phases of a vertex-program: **G**ather, **A**pply, and **S**catter. In the **gather** phase, information about adjacent vertices and edges is collected through a generalized sum over the neighborhood of the vertex u on which $Q(u)$ is run:

$$\Sigma \leftarrow \bigoplus_{v \in \mathcal{N}_u} g\left(D_u, D_{u \to v}, D_v\right). \tag{6.1}$$



(a) Twitter In-Degree

(b) Twitter Out-Degree

Figure 6.1: The in and out degree distributions of the Twitter follower network plotted in log-log scale.

where D_u, D_v, and $D_{u \to v}$ are the values (program state and meta-data) for vertices u and v and edge (u, v). The user defined sum $\oplus$ operation must be commutative and associative and can range from a numerical sum to the union of the data on all neighboring vertices and edges.

The resulting value Σ is used in the **apply** phase to update the value of the central vertex:

$$D_u^{\text{new}} \leftarrow a\left(D_u, \Sigma\right). \tag{6.2}$$

Finally the **scatter** phase uses the new value of the central vertex to update the data on adjacent edges:

$$\forall v \in \mathcal{N}_u : \quad \left(D_{u \to v}\right) \leftarrow s\left(D_u^{\text{new}}, D_{u \to v}, D_v\right). \tag{6.3}$$

The fan-in and fan-out of a vertex-program is determined by the corresponding gather and scatter phases. For instance, in PageRank, the gather phase only operates on in-edges and the scatter phase only operates on out-edges. However, for many MLDM algorithms the graph edges encode ostensibly symmetric relationships, like friendship, in which both the gather and scatter phases touch all edges. In this case the fan-in and fan-out are equal. As we will show in Section 6.3, the ability for graph parallel abstractions to support both high fan-in and fan-out computation is critical for efficient computation on natural graphs.

GraphLab and Pregel express GAS programs in very different ways. In the Pregel abstraction the gather phase is implemented using message combiners and the apply and scatter phases are expressed in the vertex program. Conversely, GraphLab exposes the entire neighborhood to the vertex-program and allows the user to define the gather and apply phases within their program. The GraphLab abstraction implicitly defines the communication aspects of the gather/scatter phases by ensuring that changes made to the vertex or edge data are automatically visible to adjacent vertices. It is also important to note that GraphLab does not differentiate between edge directions.

6.3 Challenges of Natural Graphs

The sparsity structure of natural graphs presents a unique challenge to efficient distributed graph-parallel computation. One of the hallmark properties of natural graphs is their *skewed* power-law degree distribution [Faloutsos et al., 1999]: most vertices have relatively few neighbors while a few have many neighbors (e.g., celebrities in a social network). Under a power-law degree distribution the probability that a vertex has degree d is given by:

$$\mathbf{P}(d) \propto d^{-\alpha}, \tag{6.4}$$

where the exponent α is a positive constant that controls the "skewness" of the degree distribution. Higher α implies that the graph has lower density (ratio of edges to vertices), and that the vast majority of vertices are low degree. As α decreases, the graph density and number of high degree vertices increases. Most natural graphs typically have a power-law constant around $\alpha \approx 2$. For example, Faloutsos et al. [1999] estimated that the inter-domain graph of the Internet has a power-law constant $\alpha \approx 2.2$. One can visualize the skewed power-law degree distribution by plotting the number of vertices with a given degree in log-log scale. In Figure 6.1, we plot the in and out degree distributions of the Twitter follower network demonstrating the characteristic linear power-law form.

While power-law degree distributions are empirically observable, they do not fully characterize the properties of natural graphs. While there has been substantial work (see Leskovec et al. [2007]) on more sophisticated natural graph models, the techniques in this chapter focus only on the degree distribution and do not require any other modeling assumptions.

The skewed degree distribution implies that a small fraction of the vertices are adjacent to a large fraction of the edges. For example, one percent of the vertices in the Twitter web-graph are adjacent to nearly half of the edges. This concentration of edges results in a *star-like* motif which presents challenges for existing graph-parallel abstractions:

Work Balance: The power-law degree distribution can lead to substantial work imbalance in graph parallel abstractions that treat vertices symmetrically. Since the storage, communication, and computation complexity of the Gather and Scatter phases is linear in the degree, the running time of vertex-programs can vary widely [Suri and Vassilvitskii, 2011].

Partitioning: Natural graphs are difficult to partition [Lang, 2004, Leskovec et al., 2008]. Both GraphLab and Pregel depend on graph partitioning to minimize communication and ensure work balance. However, in the case of natural graphs both are forced to resort to hash-based (random) partitioning which has extremely poor locality (Section 6.5).

Communication: The skewed degree distribution of natural-graphs leads to communication asymmetry and consequently bottlenecks. In addition, high-degree vertices can force messaging abstractions, such as Pregel, to generate and send many identical messages.

Storage: Since graph parallel abstractions must locally store the adjacency information for each vertex, each vertex requires memory linear in its degree. Consequently, high-degree vertices can exceed the memory capacity of a single machine.

Computation: While multiple vertex-programs may execute in parallel, existing graph-parallel abstractions do not parallelize *within* individual vertex-programs, limiting their scalability on high-degree vertices.

6.4 PowerGraph Abstraction

To address the challenges of computation on power-law graphs, we introduce PowerGraph, a new graph-parallel abstraction that eliminates the degree dependence of the vertex-program by directly exploiting the

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather($D_u$, $D_{u\to v}$, $D_v$) $\to$ Accum
  sum(Accum left, Accum right) $\to$ Accum
  apply($D_u$,Accum) $\to$ $D_u^{\text{new}}$
  // Run on scatter_nbrs(u)
  scatter($D_u^{\text{new}}$,$D_{u\to v}$,$D_v$) $\to$ ($D_{u\to v}^{\text{new}}$, Accum)
}
```

Figure 6.2: All PowerGraph programs must implement the stateless gather, sum, apply, and scatter functions.

GAS decomposition to factor vertex-programs over edges. By lifting the Gather and Scatter phases into the abstraction, PowerGraph is able to retain the natural "think-like-a-vertex" philosophy [Malewicz et al., 2010] while distributing the computation of a single vertex-program over the entire cluster.

PowerGraph combines the best features from both Pregel and GraphLab. From GraphLab, PowerGraph borrows the data-graph and shared-memory view of computation eliminating the need for users to architect the movement of information. From Pregel, PowerGraph borrows the commutative, associative gather concept. PowerGraph supports both the highly-parallel bulk-synchronous Pregel model of computation as well as the computationally efficient asynchronous GraphLab model of computation.

Like GraphLab, the state of a PowerGraph program factors according to a **data-graph** with user defined vertex data D_v and edge data $D_{u\to v}$. The data stored in the data-graph includes both meta-data (e.g., urls and edge weights) as well as computation state (e.g., the PageRank of vertices). In Section 6.5 we introduce vertex-cuts which allow PowerGraph to efficiently represent and store power-law graphs in a distributed environment. We now describe the PowerGraph *abstraction* and how it can be used to naturally decompose vertex-programs. Then in Section 6.5 through Section 6.7 we discuss how to implement the PowerGraph abstraction in a distributed environment.

6.4.1 GAS Vertex-Programs

Computation in the PowerGraph abstraction is encoded as a state-less **vertex-program** which implements the `GASVertexProgram` interface (Figure 6.2) and therefore explicitly factors into the gather, sum, apply, and scatter functions. Each function is invoked in stages by the PowerGraph engine following the semantics in Alg. 6.1. By factoring the vertex-program, the PowerGraph execution engine can distribute a single vertex-program over multiple machines and move computation to the data.

During the gather phase the `gather` and `sum` functions are used as a *map* and *reduce* to collect information about the neighborhood of the vertex. The `gather` function is invoked in parallel on the edges adjacent to u. The particular set of edges is determined by `gather_nbrs` which can be `none`, `in`, `out`, or `all`. The gather function is passed the data on the adjacent vertex and edge and returns a temporary accumulator (a user defined type). The result is combined using the commutative and associative `sum` operation. The final result a_u of the gather phase is passed to the apply phase and cached by PowerGraph.

After the gather phase has completed, the `apply` function takes the final accumulator and computes a new vertex value D_u which is atomically written back to the graph. The size of the accumulator a_u and

Algorithm 6.1: Vertex-Program Execution Semantics

```
Input: Center vertex $u$
if cached accumulator $a_u$ is empty then
    foreach neighbor $v$ in gather_nbrs($u$) do
        $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u \to v}, D_v))$
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach neighbor $v$ scatter_nbrs($u$) do
    $(D_{u \to v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u \to v}, D_v)$
    if $a_v$ and $\Delta a$ are not Empty then $a_v \leftarrow \text{sum}(a_v, \Delta a)$
    else $a_v \leftarrow$ Empty
```

complexity of the apply function play a central role in determining the network and storage efficiency of the PowerGraph abstraction and should be *sub-linear* and ideally constant in the degree.

During the scatter phase, the `scatter` function is invoked in parallel on the edges adjacent to u producing new edge values $D_{u \to v}$ which are written back to the data-graph. As with the gather phase, the `scatter_nbrs` determines the particular set of edges on which scatter is invoked. The scatter function returns an optional value Δa which is used to dynamically update the cached accumulator a_v for the adjacent vertex (see Section 6.4.2).

In Figure 6.3 we implement the PageRank, greedy graph coloring, and single source shortest path algorithms using the PowerGraph abstraction. In PageRank the gather and sum functions collect the total value of the adjacent vertices, the apply function computes the new PageRank, and the scatter function is used to activate adjacent vertex-programs if necessary. In graph coloring the gather and sum functions collect the set of colors on adjacent vertices, the apply function computes a new color, and the scatter function activates adjacent vertices if they violate the coloring constraint. Finally in single source shortest path (SSSP), the gather and sum functions compute the shortest path through each of the neighbors, the apply function returns the new distance, and the scatter function activates affected neighbors.

6.4.2 Delta Caching

In many cases a vertex-program will be triggered in response to a change in a *few* of its neighbors. The gather operation is then repeatedly invoked on *all* neighbors, many of which remain unchanged, thereby wasting computation cycles. For many algorithms [Ahmed et al., 2012] it is possible to dynamically maintain the result of the gather phase a_u and skip the gather on subsequent iterations.

The PowerGraph engine maintains a cache of the accumulator a_u from the previous gather phase for each vertex. The scatter function can *optionally* return an additional Δa which is atomically added to the cached accumulator a_v of the neighboring vertex v using the `sum` function. If Δa is not returned, then the neighbor's cached a_v is cleared, forcing a complete gather on the subsequent execution of the vertex-program on the vertex v. When executing the vertex-program on v the PowerGraph engine uses the cached a_v if available, bypassing the gather phase.

Intuitively, Δa acts as an additive correction on-top of the previous gather for that edge. More formally, if the accumulator type forms an **abelian group**: has a commutative and associative sum $(+)$ and an *inverse*

PageRank

```
// gather_nbrs: IN_NBRS
gather($D_u$, $D_{u\to v}$, $D_v$):
  return $D_v$.rank / #outNbrs($v$)
sum(a, b): return a + b
apply($D_u$, acc):
  rnew = 0.15 + 0.85 * acc
  $D_u$.delta = (rnew - $D_u$.rank)/
           #outNbrs(u)
  $D_u$.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter($D_u$,$D_{u\to v}$,$D_v$):
  if(|$D_u$.delta|>$\epsilon$) Activate($v$)
  return delta
```

Single Source Shortest Path (SSSP)

```
// gather_nbrs: ALL_NBRS
gather($D_u$, $D_{u\to v}$, $D_v$):
  return $D_v$ + $D_{v\to u}$
sum(a, b): return min(a, b)
apply($D_u$, new_dist):
  $D_u$ = new_dist
// scatter_nbrs: ALL_NBRS
scatter($D_u$,$D_{u\to v}$,$D_v$):
  // If changed activate neighbor
  if(changed($D_u$)) Activate($v$)
  if(increased($D_u$))
    return NULL
  else return $D_u$ + $D_{u\to v}$
```

Greedy Graph Coloring

```
// gather_nbrs: ALL_NBRS
gather($D_u$, $D_{u\to v}$, $D_v$):
  return set($D_v$)
sum(a, b): return union(a, b)
apply($D_u$, S):
  $D_u$ = min c where $c \notin S$
// scatter_nbrs: ALL_NBRS
scatter($D_u$,$D_{u\to v}$,$D_v$):
  // Nbr changed since gather
  if($D_u$ == $D_v$)
    Activate($v$)
  // Invalidate cached accum
  return NULL
```

Belief Propagation (BP)

```
// gather_nbrs: IN_NBRS
gather($D_u$, $D_{u\to v}$, $D_v$):
  swap old and new messages
  return old message
sum(a, b): return a * b
apply($D_u$, new_dist):
  update belief
// scatter_nbrs: OUT_NBRS
scatter($D_u$,$D_{u\to v}$,$D_v$):
  Recompute new out message
  if(message changed sufficiently)
    Activate($v$)
  return NULL
```

Figure 6.3: The PageRank, graph-coloring, and single source shortest path algorithms implemented in the PowerGraph abstraction. Both the PageRank and single source shortest path algorithms support delta caching in the gather phase.

$(-)$ operation, then we can define (shortening `gather` to g):

$$\Delta a = g(D_u, D_{u\to v}^{\text{new}}, D_v^{\text{new}}) - g(D_u, D_{u\to v}, D_v). \tag{6.5}$$

In the PageRank example (Figure 6.3) we take advantage of the abelian nature of the PageRank sum operation. For graph coloring the set union operation is not abelian and so we invalidate the accumulator.

6.4.3 Initiating Future Computation

The PowerGraph engine maintains a set of active vertices on which to eventually execute the vertex-program. The user initiates computation by calling `Activate(v)` or `Activate_all()`. The PowerGraph engine then proceeds to execute the vertex-program on the active vertices until none remain. Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.

Vertices can activate themselves and neighboring vertices. Each function in a vertex-program can only activate vertices visible in the arguments to that function. For example the scatter function invoked on the edge (u, v) can only activate the vertices u and v. This restriction is essential to ensure that activation events are generated on machines on which they can be efficiently processed.

The order in which activated vertices are executed is up to the PowerGraph execution engine. The only guarantee is that all activated vertices are eventually executed. This flexibility in scheduling enables PowerGraph programs to be executed both *synchronously* and *asynchronously*, leading to different tradeoffs in algorithm performance, system performance, and determinism.

Bulk Synchronous Execution

When run synchronously, the PowerGraph engine executes the gather, apply, and scatter phases in order. Each phase, called a **minor-step**, is run synchronously on all active vertices with a barrier at the end. We define a **super-step** as a complete series of GAS minor-steps. Changes made to the vertex data and edge data are committed at the end of each minor-step and are visible in the subsequent minor-step. Vertices activated in each super-step are executed in the subsequent super-step.

The synchronous execution model ensures a deterministic execution regardless of the number of machines and closely resembles Pregel. However, the frequent barriers and inability to operate on the most recent data can lead to an inefficient distributed execution and slow algorithm convergence. To address these limitations PowerGraph also supports asynchronous execution.

Asynchronous Execution

When run asynchronously, the PowerGraph engine executes active vertices as processor and network resources become available. Changes made to the vertex and edge data during the apply and scatter functions are immediately committed to the graph and visible to subsequent computation on neighboring vertices.

Unfortunately, the behavior of the asynchronous execution depends on the number machines and availability of network resources leading to non-determinism that can complicate algorithm design and debugging. Furthermore, for some algorithms, like statistical simulation (see Chapter 4), the resulting non-determinism, if not carefully controlled, can lead to instability or even divergence.

To address these challenges, GraphLab automatically enforces **serializability**: every parallel execution of vertex-programs has a corresponding sequential execution. In Chapter 5 it was shown that serializability is sufficient to support a wide range of MLDM algorithms. To achieve serializability, GraphLab prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol which requires *sequentially* grabbing locks on all neighboring vertices. Furthermore, the locking scheme used by GraphLab is unfair to high degree vertices.

PowerGraph retains the strong serializability guarantees of GraphLab while addressing its limitations. We address the problem of sequential locking by introducing a new *parallel* locking protocol (described in Section 6.7.4) which is fair to high degree vertices. In addition, the PowerGraph abstraction exposes substantially more fined grained (edge-level) parallelism allowing the entire cluster to support the execution of individual vertex programs.

6.4.4 Comparison with GraphLab / Pregel

Surprisingly, despite the strong constraints imposed by the PowerGraph abstraction, it is *possible* to emulate both GraphLab and Pregel vertex-programs in PowerGraph. To emulate a GraphLab vertex-program, we use the gather and sum functions to *concatenate* all the data on adjacent vertices and edges and then run the GraphLab program within the apply function. Similarly, to express a Pregel vertex-program, we use the gather and sum functions to combine the inbound messages (stored as edge data) and *concatenate* the list of neighbors needed to compute the outbound messages. The Pregel vertex-program then runs within the apply function generating the set of messages which are passed as vertex data to the scatter function where they are written back to the edges.

In order to address the challenges of natural graphs, the PowerGraph abstraction requires the size of the accumulator and the complexity of the apply function to be sub-linear in the degree. However, directly executing GraphLab and Pregel vertex-programs within the apply function leads the size of the accumulator and the complexity of the apply function to be *linear* in the degree eliminating many of the benefits on natural graphs.

6.5 Distributed Graph Placement

The PowerGraph abstraction relies on the distributed data-graph to store the computation state and encode the interaction between vertex-programs. The placement of the data-graph structure and data plays a central role in minimizing communication and ensuring work balance.

A common approach to placing a graph on a cluster of p machines is to construct a balanced p-way **edge-cut** (e.g., Figure 6.4a) in which vertices are evenly assigned to machines and the number of edges spanning machines is minimized. Unfortunately, the tools [Karypis and Kumar, 1998, Pellegrini and Roman, 1996] for constructing balanced edge-cuts perform poorly [Abou-Rjeili and Karypis, 2006, Lang, 2004, Leskovec et al., 2008] on power-law graphs. When the graph is difficult to partition, both GraphLab and Pregel resort to hashed (random) vertex placement. While fast and easy to implement, hashed vertex placement cuts most of the edges:

Theorem 6.5.1. *If vertices are randomly assigned to p machines then the expected fraction of edges cut is:*

$$\mathbf{E}\left[\frac{|\textit{Edges Cut}|}{|E|}\right] = 1 - \frac{1}{p}. \tag{6.6}$$

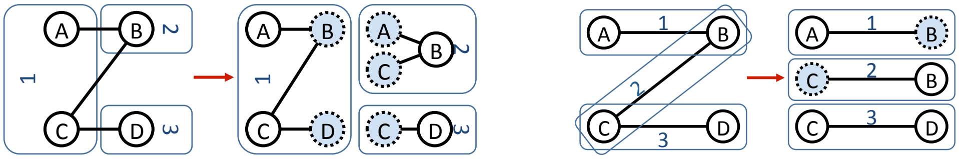
For a power-law graph with exponent α, the expected number of edges cut per-vertex is:

$$\mathbf{E}\left[\frac{|\textit{Edges Cut}|}{|V|}\right] = \left(1 - \frac{1}{p}\right)\mathbf{E}\left[\mathbf{D}[v]\right] = \left(1 - \frac{1}{p}\right)\frac{\mathbf{h}_{|V|}\left(\alpha - 1\right)}{\mathbf{h}_{|V|}\left(\alpha\right)}, \tag{6.7}$$

where the $\mathbf{h}_{|V|}\left(\alpha\right) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution.

Proof. An edge is cut if both vertices are randomly assigned to different machines. The probability that both vertices are assigned to different machines is $1 - 1/p$. □

Every cut edge contributes to storage and network overhead since both machines maintain a copy of the adjacency information and in some cases Gregor and Lumsdaine [2005], a **ghost** (local copy) of the vertex



(a) Edge-Cut

(b) Vertex-Cut

Figure 6.4: **(a)** An edge-cut and **(b)** vertex-cut of a graph into three parts. Shaded vertices are ghosts and mirrors respectively.

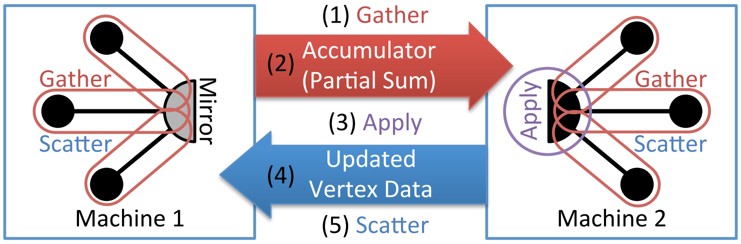


Figure 6.5: The communication pattern of the PowerGraph abstraction when using a vertex-cut. Gather function runs locally on each machine and then one accumulators is sent from each mirror to the master. The master runs the apply function and then sends the updated vertex data to all mirrors. Finally the scatter phase is run in parallel on mirrors.

and edge data. For example in Figure 6.4a we construct a three-way edge-cut of a four vertex graph resulting in five ghost vertices and all edge data being replicated. Any changes to vertex and edge data associated with a cut edge must be synchronized across the network. For example, using just two machines, a random cut will cut roughly *half* the edges, requiring $|E|/2$ communication.

6.5.1 Balanced p-way Vertex-Cut

By factoring the vertex program along the edges in the graph, The PowerGraph abstraction allows a single vertex-program to span multiple machines. In Figure 6.5 a single high degree vertex program has been split across two machines with the gather and scatter functions running in parallel on each machine and accumulator and vertex data being exchanged across the network.

Because the PowerGraph abstraction allows a single vertex-program to span multiple machines, we can improve work balance and reduce communication and storage overhead by evenly *assigning edges* to machines and allowing *vertices to span machines*. Each machine only stores the edge information for the edges assigned to that machine, evenly distributing the massive amounts of edge data. Since each edge is stored exactly once, changes to edge data do not need to be communicated. However, changes to vertex must be copied to all the machines it spans, thus the storage and network overhead depend on the number of machines spanned by each vertex.

We minimize storage and network overhead by limiting the number of machines spanned by each vertex. A balanced p-way **vertex-cut** formalizes this objective by assigning each *edge* $e \in E$ to a machine $A(e) \in \{1, \ldots, p\}$. Each vertex then spans the set of machines $A(v) \subseteq \{1, \ldots, p\}$ that contain its adjacent edges. We define the balanced vertex-cut objective:

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)| \tag{6.8}$$

$$\text{s.t.} \qquad \max_{m} |\{e \in E \mid A(e) = m\}|\,, < \lambda \frac{|E|}{p} \tag{6.9}$$

where the imbalance factor $\lambda \geq 1$ is a small constant. We use the term **replicas** of a vertex v to denote the $|A(v)|$ copies of the vertex v: each machine in $A(v)$ has a replica of v. Because changes to vertex data are communicated to all replicas, the communication overhead is also given by $|A(v)|$. The objective (Eq. (6.8)) therefore minimizes the average number of replicas in the graph and as a consequence the total storage and communication requirements of the PowerGraph engine.

For each vertex v with multiple replicas, one of the replicas is *randomly* nominated as the **master** which maintains the master version of the vertex data. All remaining replicas of v are then **mirrors** and maintain a local cached *read only* copy of the vertex data. (e.g., Figure 6.4b). For instance, in Figure 6.4b we construct a three-way vertex-cut of a graph yielding only 2 mirrors. Any changes to the vertex data (e.g., the Apply function) must be made to the master which is then immediately replicated to all mirrors.

Vertex-cuts address the major issues associated with edge-cuts in power-law graphs. Percolation theory [Albert et al., 2000] suggests that power-law graphs have good vertex-cuts. Intuitively, by cutting a small fraction of the very high degree vertices we can quickly shatter a graph. Furthermore, because the balance constraint (Eq. (6.9)) ensures that edges are uniformly distributed over machines, we naturally achieve improved work balance even in the presence of very high-degree vertices.

The simplest method to construct a vertex cut is to randomly assign edges to machines. Random (hashed) edge placement is fully data-parallel, achieves nearly perfect balance on large graphs, and can be applied in the streaming setting. In the following theorem, we relate the expected normalized replication factor (Eq. (6.8)) to the number of machines and the power-law constant α.

Theorem 6.5.2 (Randomized Vertex Cuts)**.** *A random vertex-cut on p machines has an expected replication:*

$$\mathbf{E}\left[\frac{1}{|V|} \sum_{v \in V} |A(v)|\right] = \frac{p}{|V|} \sum_{v \in V} \left(1 - \left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}\right). \tag{6.10}$$

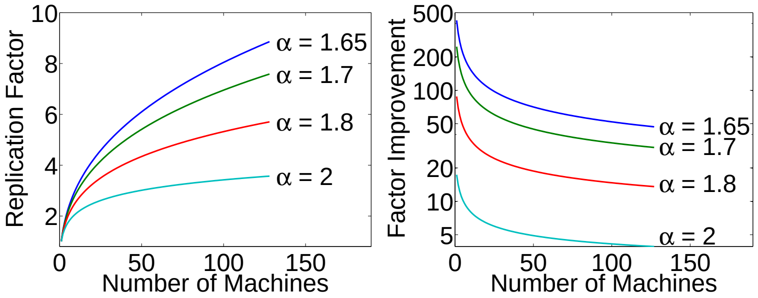
where $\mathbf{D}[v]$ denotes the degree of vertex v. For a power-law graph the expected replication (Figure 6.6a) is determined entirely by the power-law constant α:

$$\mathbf{E}\left[\frac{1}{|V|} \sum_{v \in V} |A(v)|\right] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)} \sum_{d=1}^{|V|-1} \left(\frac{p-1}{p}\right)^{d} d^{-\alpha}, \tag{6.11}$$

where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution.

Proof. By linearity of expectation:

$$\mathbf{E}\left[\frac{1}{|V|} \sum_{v \in V} |A(v)|\right] = \frac{1}{|V|} \sum_{v \in V} \mathbf{E}\left[|A(v)|\right], \tag{6.12}$$



(a) V-Sep. Bound (b) V-Sep. Improvement

Figure 6.6: **(a)** Expected replication factor for different power-law constants. **(b)** The ratio of the expected communication and storage cost of random edge cuts to random vertex cuts as a function of the number machines. This graph assumes that edge data and vertex data are the same size.

The expected replication $\mathbf{E}\left[|A(v)|\right]$ of a single vertex v can be computed by considering the process of randomly assigning the $\mathbf{D}[v]$ edges adjacent to v. Let the indicator X_i denote the event that vertex v has at least one of its edges on machine i. The expectation $\mathbf{E}\left[X_i\right]$ is then:

$$\begin{aligned} \mathbf{E}\left[X_i\right] &= 1 - \mathbf{P}\left(v \text{ has no edges on machine } i\right) && (6.13) \\ &= 1 - \left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}, && (6.14) \end{aligned}$$

The expected replication factor for vertex v is then:

$$\mathbf{E}\left[|A(v)|\right] = \sum_{i=1}^{p} \mathbf{E}\left[X_i\right] = p\left(1 - \left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}\right). \quad (6.15)$$

Treating $\mathbf{D}[v]$ as a Zipf random variable:

$$\mathbf{E}\left[\frac{1}{|V|}\sum_{v \in V} |A(v)|\right] = \frac{p}{|V|}\sum_{v \in V}\left(1 - \mathbf{E}\left[\left(\frac{p-1}{p}\right)^{\mathbf{D}[v]}\right]\right), \quad (6.16)$$

and taking the expectation under $\mathbf{P}\left(d\right) = d^{-\alpha} / \, \mathbf{h}_{|V|}\left(\alpha\right)$:

$$\mathbf{E}\left[\left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}\right] = \frac{1}{\mathbf{h}_{|V|}\left(\alpha\right)} \sum_{d=1}^{|V|-1} \left(1 - \frac{1}{p}\right)^{d} d^{-\alpha}. \quad (6.17)$$

□

While lower α values (more high-degree vertices) imply a higher replication factor (Figure 6.6a) the effective gains of vertex-cuts relative to edge cuts (Figure 6.6b) actually *increase* with lower α. In Figure 6.6b we plot the ratio of the expected costs (comm. and storage) of random edge-cuts (Eq. (6.7)) to the expected costs of random vertex-cuts (Eq. (6.11)) demonstrating order of magnitude gains.

Finally, the vertex cut model is also highly effective for regular graphs since in the event that a good edge-cut can be found it can be converted to a better vertex cut:

Theorem 6.5.3. *For a given an edge-cut with g ghosts, **any** vertex cut along the same partition boundary has strictly fewer than g mirrors.*

Proof of Theorem 6.5.3. Consider the two-way edge cut which cuts the set of edges $E' \in E$ and let V' be the set of vertices in E'. The total number of ghosts induced by this edge partition is therefore $|V'|$. If we then select and delete arbitrary vertices from V' along with their adjacent edges until no edges remain, then the set of deleted vertices corresponds to a vertex-cut in the original graph. Since at most $|V'| - 1$ vertices may be deleted, there can be at most $|V'| - 1$ mirrors. □

6.5.2 Greedy Vertex-Cuts

We can improve upon the randomly constructed vertex-cut by de-randomizing the edge-placement process. The resulting algorithm is a sequential greedy heuristic which places the next edge on the machine that minimizes the conditional expected replication factor. To construct the de-randomization we consider the task of placing the $i + 1$ edge after having placed the previous i edges. Using the conditional expectation we define the objective:

$$\arg\min_k \mathbf{E}\left[\sum_{v \in V} |A(v)| \mid A_i, A(e_{i+1}) = k\right], \tag{6.18}$$

where A_i is the assignment for the previous i edges. Using Theorem 6.5.2 to evaluate Eq. (6.18) we obtain the following edge placement rules for the edge (u, v):

Case 1: If $A(u)$ and $A(v)$ intersect, then the edge should be assigned to a machine in the intersection.

Case 2: If $A(u)$ and $A(v)$ are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

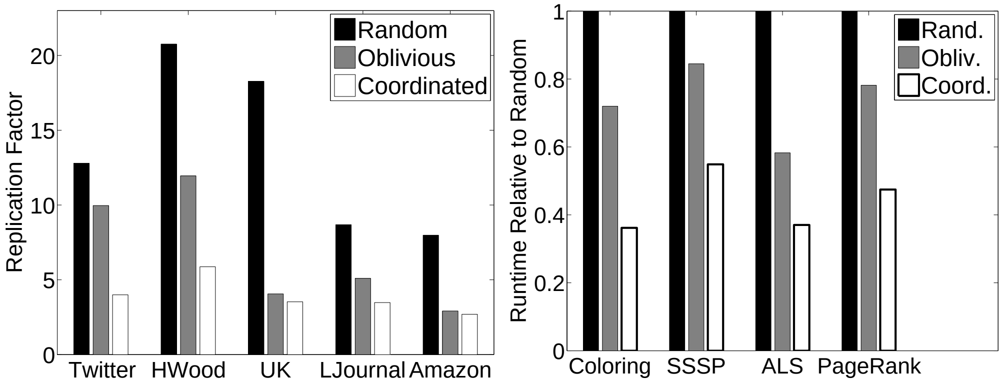
Case 4: If neither vertex has been assigned, then assign the edge to the least loaded machine.

Because the greedy-heuristic is a de-randomization it is guaranteed to obtain an expected replication factor that is no worse than random placement and in practice can be much better. Unlike the randomized algorithm, which is embarrassingly parallel and easily distributed, the greedy algorithm requires coordination between machines. We consider two distributed implementations:

Coordinated: maintains the values of $A_i(v)$ in a distributed table. Then each machine runs the greedy heuristic and periodically updates the distributed table. Local caching is used to reduce communication at the expense of accuracy in the estimate of $A_i(v)$.

Oblivious: runs the greedy heuristic independently on each machine. Each machine maintains its own estimate of A_i with no additional communication.

In Figure 6.8a, we compare the replication factor of both heuristics against random vertex cuts on the Twitter follower network. We plot the replication factor as a function of the number of machines (EC2 instances described in Section 6.7) and find that random vertex cuts match the predicted replication given in Theorem 6.5.2. Furthermore, the greedy heuristics substantially improve upon random placement with an order of magnitude reduction in the replication factor, and therefore communication and storage costs. For a fixed number of machines ($p = 32$), we evaluated (Figure 6.7a) the replication factor of the two heuristics on five real-world graphs (Table 6.1a). In all cases the greedy heuristics out-perform random placement,



(a) Actual Replication

(b) Effect of Partitioning

Figure 6.7: **(a)** The actual replication factor on 32 machines. **(b)** The effect of partitioning on runtime.

Graph	$\vert V\vert$	$\vert E\vert$
Twitter [Kwak et al., 2010]	41M	1.4B
UK [Bordino et al., 2008]	132.8M	5.5B
Amazon [Boldi and Vigna, 2004, Boldi et al., 2011]	0.7M	5.2M
LiveJournal [Leskovec, 2011]	5.4M	79M
Hollywood [Boldi and Vigna, 2004, Boldi et al., 2011]	2.2M	229M

(a) Real world graphs

α	**# Edges**
1.8	641,383,778
1.9	245,040,680
2.0	102,838,432
2.1	57,134,471
2.2	35,001,696

(b) Synthetic Graphs

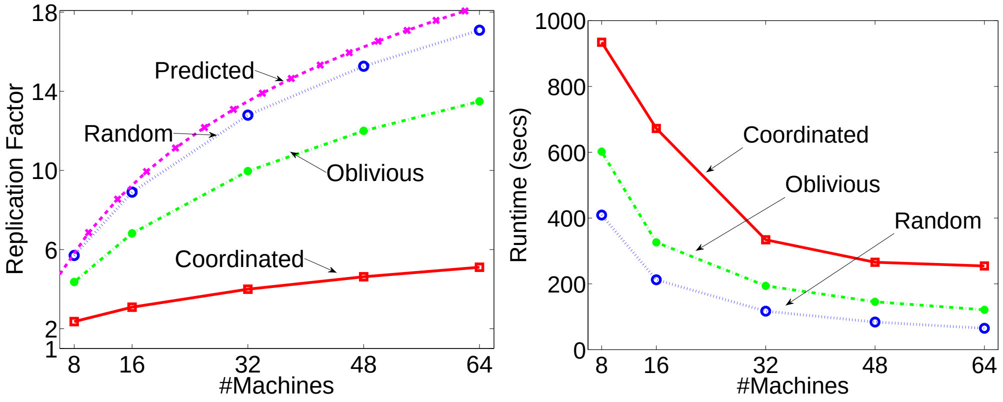
Table 6.1: **(a)** A collection of Real world graphs. **(b)** Randomly constructed ten-million vertex power-law graphs with varying α. Smaller α produces denser graphs.

while doubling the load time (Figure 6.8b). The Oblivious heuristic achieves compromise by obtaining a relatively low replication factor while only slightly increasing runtime.

6.6 Abstraction Comparison

In this section, we experimentally characterize the dependence on α and the relationship between fan-in and fan-out by using the Pregel, GraphLab, and PowerGraph abstractions to run PageRank on five synthetically constructed power-law graphs. Each graph has ten-million vertices and an α ranging from 1.8 to 2.2. The graphs were constructed by randomly sampling the out-degree of each vertex from a Zipf distribution and then adding out-edges such that the in-degree of each vertex is nearly identical. We then inverted each graph to obtain the corresponding power-law fan-in graph. The density of each power-law graph is determined by α and therefore each graph has a different number of edges (see Table 6.1b).

We used the GraphLab v1 C++ implementation from Chapter 5 and added instrumentation to track network usage. As of the writing of this chapter, public implementations of Pregel (e.g., Giraph) were unable to handle even our smaller synthetic problems due to memory limitations. Consequently, we used Piccolo [Power and Li, 2010] as a proxy implementation of Pregel since Piccolo naturally expresses the Pregel abstraction and provides an efficient C++ implementation with dynamic load-balancing. Finally, we used our implementation of PowerGraph described in Section 6.7.



(a) Replication Factor (Twitter)

(b) Ingress time (Twitter)

Figure 6.8: **(a,b)** Replication factor and runtime of graph ingress for the Twitter follower network as a function of the number of machines for random, oblivious, and coordinated vertex-cuts.

All experiments in this section are evaluated on an eight node Linux cluster. Each node consists of two quad-core Intel Xeon E5620 processors with 32 GB of RAM and is connected via 1-GigE Ethernet. All systems were compiled with GCC 4.4. GraphLab and Piccolo used random edge-cuts while PowerGraph used random vertex-cuts.. Results are averaged over 20 iterations.

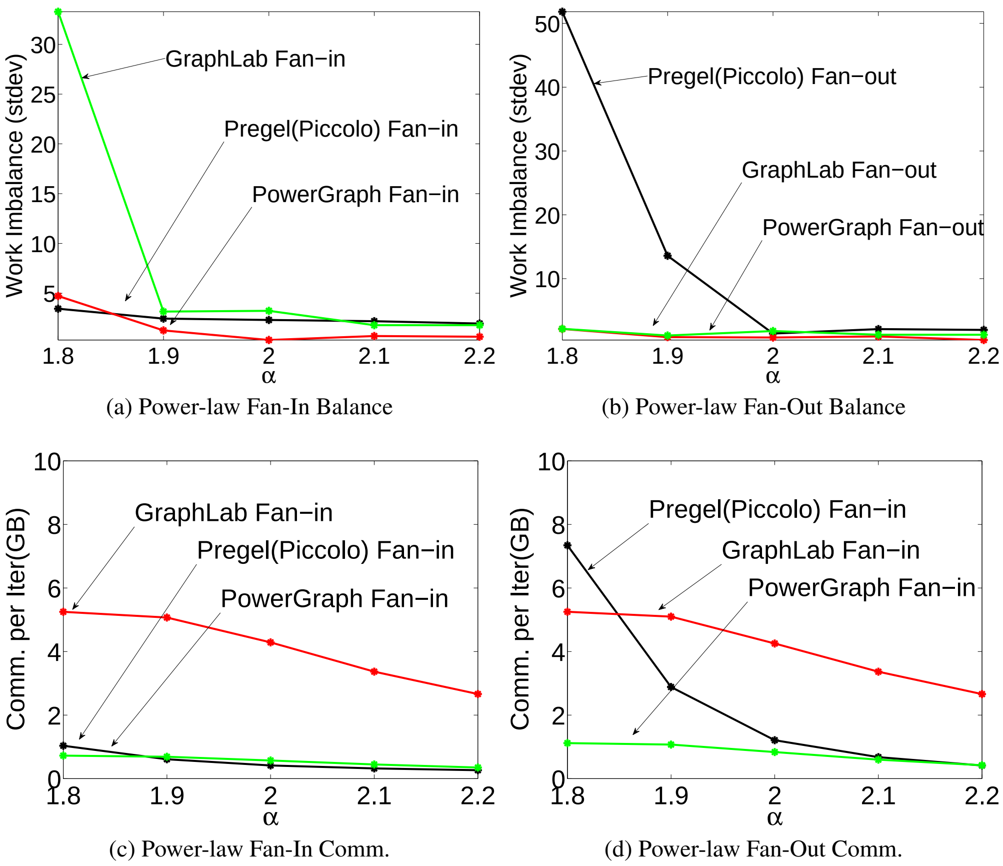
6.6.1 Computation Imbalance

The *sequential* component of the PageRank vertex-program is proportional to out-degree in the Pregel abstraction and in-degree in the GraphLab abstraction. Alternatively, PowerGraph eliminates this sequential dependence by distributing the computation of *individual* vertex-programs over multiple machines. Therefore we expect, highly-skewed (low α) power-law graphs to increase work imbalance under the Pregel (fan-in) and GraphLab (fan-out) abstractions but not under the PowerGraph abstraction, which evenly distributed high-degree vertex-programs. To evaluate this hypothesis we ran eight "workers" per system (64 total workers) and recorded the vertex-program time on each worker.

In Figure 6.9a and Figure 6.9b we plot the *standard deviation* of worker per-iteration runtimes, a measure of work imbalance, for power-law fan-in and fan-out graphs respectively. Higher standard deviation implies greater imbalance. While lower α increases work imbalance for GraphLab (on fan-in) and Pregel (on fan-out), the PowerGraph abstraction is unaffected in either edge direction.

6.6.2 Communication Imbalance

Because GraphLab and Pregel use edge-cuts, their communication volume is proportional to the number of ghosts: the replicated vertex and edge data along the partition boundary. If one message is sent per edge, Pregel's combiners ensure that exactly one network message is transmitted for each ghost. Similarly, at the end of each iteration GraphLab synchronizes each ghost and thus the communication volume is also proportional to the number of ghosts. PowerGraph on the other hand uses vertex-cuts and only synchronizes mirrors after each iteration. The communication volume of a complete iteration is therefore proportional to



(a) Power-law Fan-In Balance

(b) Power-law Fan-Out Balance

(c) Power-law Fan-In Comm.

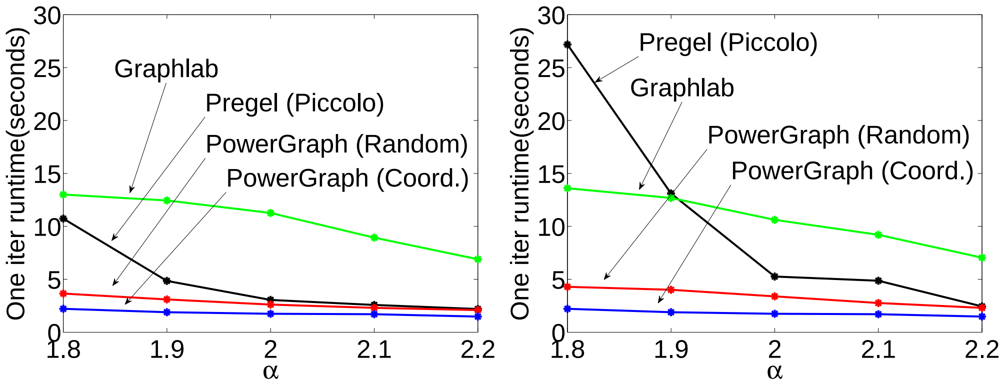
(d) Power-law Fan-Out Comm.

Figure 6.9: **Synthetic Experiments: Work Imbalance and Communication. (a, b)** Standard deviation of worker computation time across 8 distributed workers for each abstraction on power-law fan-in and fan-out graphs. **(b, c)** Bytes communicated per iteration for each abstraction on power-law fan-in and fan-out graphs.

the number of mirrors induced by the vertex-cut. As a consequence we expect that PowerGraph will reduce communication volume.

In Figure 6.9c and Figure 6.9d we plot the bytes communicated per iteration for all three systems under power-law fan-in and fan-out graphs. Because Pregel only sends messages along out-edges, Pregel communicates more on power-law fan-out graphs than on power-law fan-in graphs.

On the other hand, GraphLab and PowerGraph's communication volume is invariant to power-law fan-in and fan-out since neither considers edge direction during data-synchronization. However, PowerGraph communicates significantly less than GraphLab which is a direct result of the efficacy of vertex cuts. Finally, PowerGraph's total communication increases only marginally on the denser graphs and is the lowest overall.



(a) Power-law Fan-In Runtime

(b) Power-law Fan-Out Runtime

Figure 6.10: **Synthetic Experiments Runtime.** **(a, b)** Per iteration runtime of each abstraction on synthetic power-law graphs.

6.6.3 Runtime Comparison

PowerGraph significantly out-performs GraphLab and Pregel on low α graphs. In Figure 6.10a and Figure 6.10b we plot the per iteration runtime for each abstraction. In both cases the overall runtime performance closely matches the communication overhead (Figure 6.9c and Figure 6.9d) while the computation imbalance (Figure 6.9a and Figure 6.9b) appears to have little effect. The limited effect of imbalance is due to the relatively lightweight nature of the PageRank computation and we expect more complex algorithms (e.g., statistical inference) to be more susceptible to imbalance. However, when greedy (coordinated) partitioning is used we see an additional 25% to 50% improvement in runtime.

6.7 Implementation and Evaluation

In this section, we describe and evaluate our implementation of the PowerGraph[1] system. All experiments are performed on a 64 node cluster of Amazon EC2 `cc1.4xlarge` Linux instances. Each instance has two quad core Intel Xeon X5570 processors with 23GB of RAM, and is connected via 10 GigE Ethernet. PowerGraph was written in C++ and compiled with GCC 4.5.

We implemented three variations of the PowerGraph abstraction. To demonstrate their relative implementation complexity, we provide the line counts, excluding common support code:

Bulk Synchronous (Sync): A fully synchronous implementation of PowerGraph as described in Section 6.4.3. [600 lines]

Asynchronous (Async): An asynchronous implementation of PowerGraph which allows arbitrary interleaving of vertex-programs Section 6.4.3. [900 lines]

Asynchronous Serializable (Async+S): An asynchronous implementation of PowerGraph which guarantees serializability of *all* vertex-programs (equivalent to "edge consistency" in GraphLab). [1600 lines]

[1] PowerGraph is now part of the GraphLab2 open source project and can be obtained from `http://graphlab.org`.

In all cases the system is entirely symmetric with no single coordinating instance or scheduler. Each instances is given the list of other machines and start by reading a unique subset of the graph data files from HDFS. TCP connections are opened with other machines as needed to build the distributed graph and run the engine.

6.7.1 Graph Loading and Placement

The graph structure and data are loaded from a collection of text files stored in a distributed file-system (HDFS) by all instances in parallel. Each machine loads a separate subset of files (determined by hashing) and applies one of the three distributed graph partitioning algorithms to place the data *as it is loaded*. As a consequence partitioning is accomplished in parallel and data is immediately placed in its final location. Unless specified, all experiments were performed using the oblivious algorithm. Once computation is complete, the final vertex and edge data are saved back to the distributed file-system in parallel.

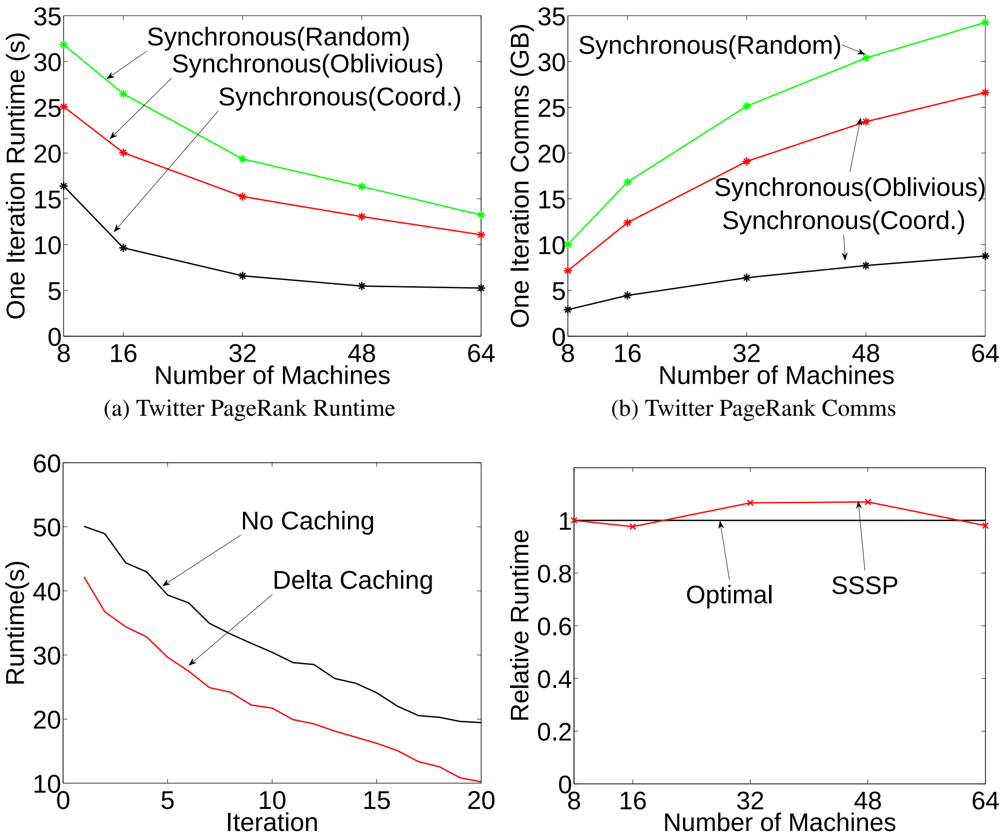
In Figure 6.7b, we evaluate the performance of a collection of algorithms varying the partitioning procedure. Our simple partitioning heuristics are able to improve performance significantly across *all algorithms*, decreasing runtime and memory utilization. Furthermore, the runtime scales *linearly* with the replication factor: halving the replication factor approximately halves runtime.

6.7.2 Synchronous Engine (Sync)

Our synchronous implementation closely follows the description in Section 6.4.3. Each machine runs a single multi-threaded instance to maximally utilize the multi-core architecture. We rely on background communication to achieve computation/communication interleaving. The synchronous engine's fully deterministic execution makes it easy to reason about programmatically and minimizes effort needed for tuning and performance optimizations.

In Figure 6.11a and Figure 6.11b we plot the runtime and total communication of one iteration of PageRank on the Twitter follower network for each partitioning method. To provide a point of comparison (Table 6.2), the Spark [Zaharia et al., 2010] framework computes one iteration of PageRank on the same graph in 97.4s on a 50 node-100 core cluster [Stanton and Kliot, 2011]. PowerGraph is therefore between 3-8x faster than Spark on a comparable number of cores. On the full cluster of 512 cores, we can compute one iteration in 3.6s.

The greedy partitioning heuristics improves both performance and scalability of the engine at the cost of increased load-time. The load time for random, oblivious, and coordinated placement were 59, 105, and 239 seconds respectively. While greedy partitioning heuristics increased load-time by up to a factor of four, they still improve overall runtime if more than 20 iterations of PageRank are performed. In Figure 6.11c we plot the runtime of each iteration of PageRank on the Twitter follower network. Delta caching improves performance by avoiding unnecessary gather computation, decreasing total runtime by 45%. Finally, in Figure 6.11d we evaluate weak-scaling: ability to scale while keeping the problem size per processor constant. We run SSSP (Figure 6.3) on synthetic power-law graphs ($\alpha = 2$), with ten-million vertices per machine. Our implementation demonstrates nearly optimal weak-scaling and requires only 65s to solve a 6.4B edge graph.



(a) Twitter PageRank Runtime

(b) Twitter PageRank Comms

(c) Twitter PageRank Delta Cache

(d) SSSP Weak Scaling

Figure 6.11: **Synchronous Experiments (a,b)** Synchronous PageRank Scaling on Twitter graph. **(c)** The PageRank per iteration runtime on the Twitter graph with and without delta caching. **(d)** Weak scaling of SSSP on synthetic graphs.

6.7.3 Asynchronous Engine (Async)

We implemented the asynchronous PowerGraph execution model (Section 6.4.3) using a simple state machine for each vertex which can be either: `INACTIVE`, `GATHER`, `APPLY` or `SCATTER`. Once activated, a vertex enters the gathering state and is placed in a *local* scheduler which assigns cores to active vertices allowing many vertex-programs to run simultaneously thereby hiding communication latency. While arbitrary interleaving of vertex programs is permitted, we avoid data races by ensuring that individual gather, apply, and scatter calls have exclusive access to their arguments.

We evaluate the performance of the Async engine by running PageRank on the Twitter follower network. In Figure 6.12a, we plot throughput (number of vertex-program operations per second) against the number of machines. Throughput increases moderately with both the number of machines as well as improved partitioning. We evaluate the gains associated with delta caching (Section 6.4.2) by measuring throughput as a function of time (Figure 6.12b) with caching enabled and with caching disabled. Caching allows the algorithm to converge faster with fewer operations. Surprisingly, when caching is disabled, the throughput increases over time. Further analysis reveals that the computation gradually focuses on high-degree vertices, increasing the computation/communication ratio.

We evaluate the graph coloring vertex-program (Figure 6.3) which cannot be run synchronously since all vertices would change to the same color on every iteration. Graph coloring is a proxy for many MLDM algorithms like Gibbs sampling (Chapter 4). In Figure 6.12c we evaluate weak-scaling on synthetic power-law graphs ($\alpha = 2$) with five-million vertices per machine and find that the Async engine performs nearly optimally. The slight increase in runtime may be attributed to an increase in the number of colors due to increasing graph size.

6.7.4 Async. Serializable Engine (Async+S)

The Async engine is useful for a broad range of tasks, providing high throughput and performance. However, unlike the synchronous engine, the asynchronous engine is difficult to reason about programmatically. We therefore extended the Async engine to enforce serializability.

The Async+S engine ensures serializability by preventing adjacent vertex-programs from running simultaneously. Ensuring serializability for graph-parallel computation is equivalent to solving the dining philosophers problem where each vertex is a philosopher, and each edge is a fork. GraphLab implemented the Dijkstra [1971] algorithm where forks are acquired *sequentially* according to a total ordering. Instead, PowerGraph implements the Chandy and Misra [1984] solution which acquires all forks simultaneously, permitting a high degree of parallelism. We extend the Chandy and Misra [1984] solution to the vertex-cut setting by enabling each vertex replica to request only forks for local edges and using a simple consensus protocol to establish when all replicas have succeeded.

We evaluate the scalability and computational efficiency of the Async+S engine on the graph coloring task. We observe in Figure 6.12c that the amount of achieved parallelism does not increase linearly with the number of vertices. Because the density (i.e., contention) of power-law graphs increases super-linearly with the number of vertices, we do not expect the amount of serializable parallelism to increase linearly.

In Figure 6.12d, we plot the proportion of edges that satisfy the coloring condition (both vertices have different colors) for both the Async and the Async+S engines. While the Async engine quickly satisfies the coloring condition for most edges, the remaining 1% take 34% of the runtime. We attribute this behavior to frequent races on tightly connected vertices. Alternatively, the Async+S engine performs more uniformly. If we examine the total number of user operations we find that the Async engine does more than *twice* the work of the Async+S engine.

Finally, we evaluate the Async and the Async+S engines on a popular machine learning algorithm: Alternating Least Squares (ALS). The ALS algorithm has a number of variations which allow it to be used in a wide range of applications including user personalization [Zhou et al., 2008] and document semantic analysis [Hofmann, 1999]. We apply ALS to the Wikipedia term-document graph consisting of 11M vertices and 315M edges to extract a mixture of topics representation for each document and term. The number of topics d is a free parameter that determines the computational complexity $O\left(d^3\right)$ of each vertex-program. In Figure 6.13a, we plot the ALS throughput on the Async engine and the Async+S engine. While the throughput of the Async engine is greater, the gap between engines shrinks as d increases and computation dominates the consistency overhead. To demonstrate the importance of serializability, we plot in Figure 6.13b the training error, a measure of solution quality, for both engines. We observe that while the Async engine has greater throughput, the Async+S engine *converges* faster.

The complexity of the Async+S engine is justified by the necessity for serializability in many applications (e.g., ALS). Furthermore, serializability adds predictability to the nondeterministic asynchronous execution.

PageRank	Runtime	$\lvert V \rvert$	$\lvert E \rvert$	System
Kang et al. [2009]	198s	–	1.1B	50x8
Zaharia et al. [2010]	97.4s	40M	1.5B	50x2
Ekanayake et al. [2010]	36s	50M	1.4B	64x4
PowerGraph (Sync)	3.6s	40M	1.5B	64x8

Triangle Count	Runtime	$\lvert V \rvert$	$\lvert E \rvert$	System
Suri and Vassilvitskii [2011]	423m	40M	1.4B	1636x?
PowerGraph (Sync)	1.5m	40M	1.4B	64x16

LDA	Tok/sec	Topics	System
Smola and Narayanamurthy [2010]	150M	1000	100x8
PowerGraph (Async)	110M	1000	64x16

Table 6.2: Relative performance of PageRank, triangle counting, and LDA on similar graphs. PageRank runtime is measured per iteration. Both PageRank and triangle counting were run on the Twitter follower network and LDA was run on Wikipedia. The systems are reported as number of nodes by number of cores.

For example, even graph coloring may not terminate on dense graphs unless serializability is ensured.

6.7.5 Fault Tolerance

Like GraphLab and Pregel, PowerGraph achieves fault-tolerance by saving a snapshot of the data-graph. The synchronous PowerGraph engine constructs the snapshot between super-steps and the asynchronous engine suspends execution to construct the snapshot. An asynchronous snapshot using GraphLab's snapshot algorithm can also be implemented. The checkpoint overhead, typically a few seconds for the largest graphs we considered, is small relative to the running time of each application.

6.7.6 MLDM Applications

To assess the relative performance of PowerGraph we compare against published results on several real-world problems and datasets. In Table 6.2 we provide comparisons of the PowerGraph system with published results on similar data for PageRank, Triangle Counting [Suri and Vassilvitskii, 2011], and collapsed Gibbs sampling for the LDA model [Smola and Narayanamurthy, 2010].

We compared our PageRank per iteration runtime with published results on other systems and were able to achieve order of magnitude gains compared to published results. However, to provide a fair comparison we did not take advantage of *dynamic* or *asynchronous* scheduling which would require accuracy information for the published results. In Figure 6.14a we compare the PageRank algorithm when run synchronously, asynchronously, and asynchronously with dynamic scheduling (following the GrAD methodology). The dynamic asynchronous scheduling substantially accelerates convergence (reduction in error). To get a sense of the importance of dynamic scheduling we also plot the number of vertices against the update count in Figure 6.14b. More than 50% of the vertices are updated only *once*.

The triangle counting application computes for each vertex the number of neighbors that are also connected (form a triangle). Triangle counting is used to estimate the strength of the community around a vertex. Users with a greater triangles to neighbors ratio have a tighter community. Triangle counting was implemented in the PowerGraph abstraction by using the gather phase to collect the set of neighboring vertices and the scatter phase to compute and store the intersection of the neighborhood set on each edge. In contrast, Suri and Vassilvitskii [2011] used Hadoop and were forced to emit a large number of intermediate key-value pairs (high fan-out) flooding the network. As a consequence PowerGraph is over two orders of magnitude faster.

For LDA, the state-of-the-art solution is a heavily optimized system designed for this specific task by Smola and Narayanamurthy [2010]. We implemented an asynchronous version of the collapsed Gibbs sampler for LDA similar to the one described by Smola and Narayanamurthy [2010]. Neither samplers are ergodic but both generate reasonable topics relatively quickly. The PowerGraph sampler enforces stronger local consistency guarantees and could potentially converge faster for strongly correlated documents. Nonetheless, we compared both samplers in terms of the sampling throughput. We find that PowerGraph is able to achieve comparable performance using only 200 lines of user code.

6.8 Additional Related Work

The vertex-cut approach to distributed graph placement is related to work [Catalyurek and Aykanat, 1996, Devine et al., 2006] in hypergraph partitioning. In particular, a vertex-cut problem can be cast as a hypergraph-cut problem by converting each edge to a vertex, and each vertex to a hyper-edge. However, existing hypergraph partitioning can be very time intensive. While our cut objective is similar to the "communication volume" objective, the streaming vertex cut setting described in this chapter is novel. Stanton and Kliot [2011] developed several heuristics for the streaming edge-cuts but do not consider the vertex-cut problem.

Several [Buluç and Gilbert, 2011, Kang et al., 2009] have proposed generalized sparse matrix vector multiplication as a basis for graph-parallel computation. These abstractions operate on commutative associative semi-rings and therefore also have generalized gather and sum operations. However, they do not support the more general apply and scatter operations, as well as mutable edge-data and are based on a strictly synchronous model in which all computation is run in every iteration.

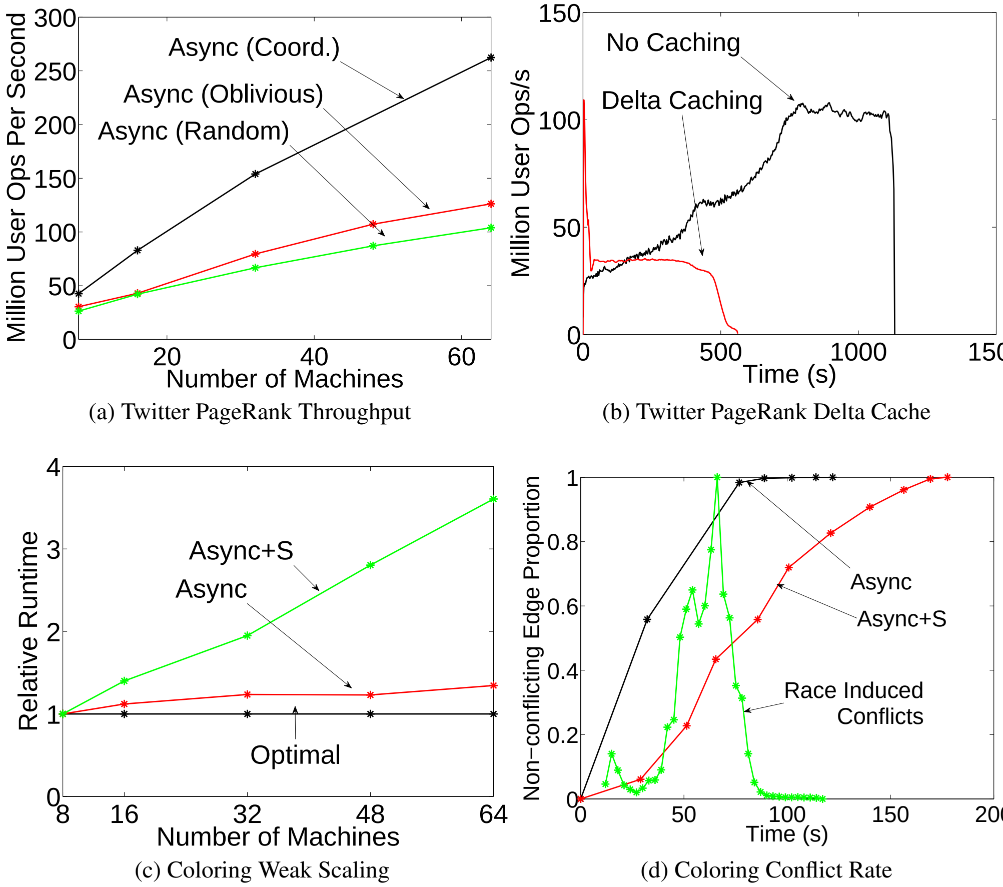
While we discuss Pregel and GraphLab in detail, there are other similar graph-parallel abstractions. Closely related to Pregel is BPGL [Gregor and Lumsdaine, 2005] which implements a synchronous traveler model. Signal/Collect [Stutz et al., 2010] is an asynchronous messaging abstraction with dynamic scheduling. However signal/collect is a shared memory abstraction. Alternatively, Kineograph [Cheng et al., 2012] presents a graph-parallel framework for time-evolving graphs which mixes features from both GraphLab and Piccolo. Pujol et al. [2010] present a distributed graph database but do not explicitly consider the power-law structure. Finally, Kyrola et al. [2012] present GraphChi: an efficient single-machine disk-based implementation of the GraphLab abstraction. Impressively, it is able to significantly out-perform large Hadoop deployments on many graph problems while using only a single machine: performing one iteration of PageRank on the Twitter Graph in only 158s (PowerGraph: 3.6s). The techniques described in GraphChi can be used to add out-of-core storage to PowerGraph.

6.9 Conclusion

The need to reason about large-scale graph-structured data has driven the development of new graph-parallel abstractions such as GraphLab and Pregel. However graphs derived from real-world phenomena often exhibit power-law degree distributions, which are difficult to partition and can lead to work imbalance and substantially increased communication and storage.

To address these challenges, we introduced the PowerGraph abstraction which exploits the **G**ather-**A**pply-**S**catter model of computation to factor vertex-programs over edges, splitting high-degree vertices and exposing greater parallelism in natural graphs. We then introduced vertex-cuts and a collection of fast greedy heuristics to substantially reduce the storage and communication costs of large distributed power-law graphs. We theoretically related the power-law constant to the communication and storage requirements of the PowerGraph system and empirically evaluate our analysis by comparing against GraphLab and Pregel. Finally, we evaluate the PowerGraph system on several large-scale problems using a 64 node EC2 cluster and demonstrating the scalability and efficiency and in many cases order of magnitude gains over published results.

We are actively using PowerGraph to explore new large-scale machine learning algorithms. We are beginning to study how vertex replication and data-dependencies can be used to support fault-tolerance without check-pointing. In addition, we are exploring ways to support time-evolving graph structures. Finally, we believe that many of the core ideas in the PowerGraph abstraction can have a significant impact in the design and implementation of graph-parallel systems beyond PowerGraph.



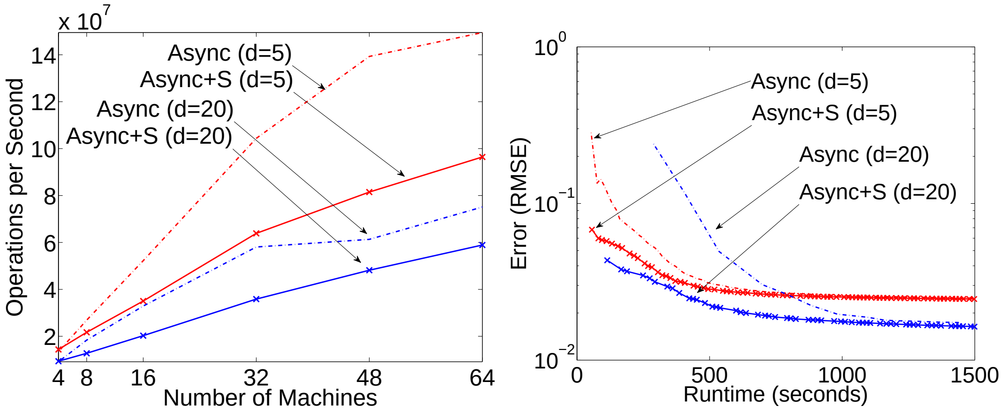
(a) Twitter PageRank Throughput

(b) Twitter PageRank Delta Cache

(c) Coloring Weak Scaling

(d) Coloring Conflict Rate

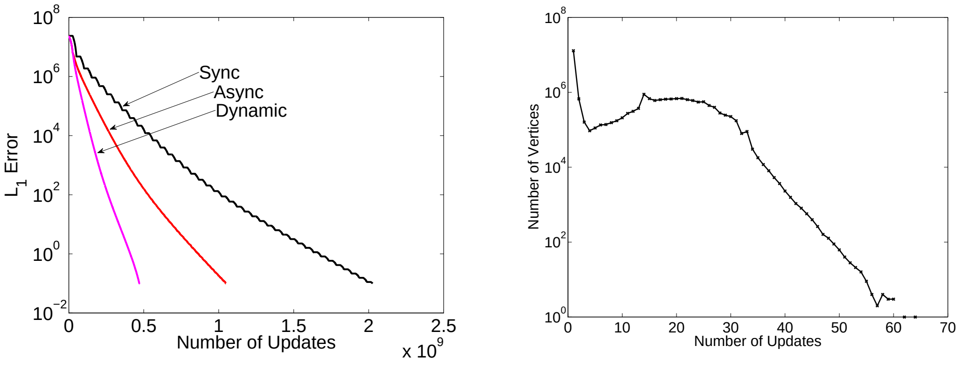
Figure 6.12: **Asynchronous Experiments (a)** Number of user operations (gather/apply/scatter) issued per second by Dynamic PageRank as # machines is increased. **(b)** Total number of user ops with and without caching plotted against time. **(c)** Weak scaling of the graph coloring task using the Async engine and the Async+S engine **(d)** Proportion of non-conflicting edges across time on a 8 machine, 40M vertex instance of the problem. The green line is the rate of conflicting edges introduced by the lack of consistency (peak 236K edges per second) in the Async engine. When the Async+S engine is used no conflicting edges are ever introduced.



(a) ALS Throughput

(b) ALS Convergence

Figure 6.13: **(a)** The throughput of ALS measured in millions of User Operations per second. **(b)** Training error (lower is better) as a function of running time for ALS application.



(a) Page Rank Update Counts

(b) Page Rank Update Counts

Figure 6.14: **PageRank Updates and Convergence:** To assess the gains of applying the GrAD methodology to PageRank we compare synchronous, asynchronous, and asynchronous + dynamic implementations achieved by using the synchronous and asynchronous engine. We plot (in log-scale) the L_1 error **(a)** relative to the true PageRank (computed by running synchronous PageRank to convergence) as a function of vertex updates. To get a sense of the importance of *dynamic* scheduling we plot **(b)** the number of vertices against the number of times those vertices were updated. More than 50% of the vertices are only updated *once*.

Chapter 7

Conclusion and Observations

Probabilistic reasoning lies at the foundation of machine learning and statistics and is essential to extracting signal from noisy data. This thesis explored the task of designing parallel and distributed algorithms and systems for probabilistic reasoning. We proposed the three design principles:

1. **Graphical**: Explicitly represent statistical and computational dependencies as a sparse graph.
2. **Asynchronous**: Allow computation to proceed asynchronously as dependencies are satisfied obeying the dependency structure encoded in the graph.
3. **Dynamic**: Adaptively identify and prioritize sub-problems that when solved make the most progress towards the solution.

The **Gr**aphical, **A**synchronous, and **D**ynamic design principles form the basis of the **GrAD** design methodology which lead to the thesis statement:

> By factoring large-scale probabilistic computation with respect to the underlying *graphical structure*, *asynchronously* addressing sequential dependencies, and *dynamically* scheduling computation we can construct efficient parallel and distributed systems capable of scalable probabilistic reasoning.

We evaluated the thesis statement by first developing a series of efficient parallel and distributed probabilistic inference algorithms. In Chapter 3 we studied parallel and distributed algorithms for loopy belief propagation. We demonstrated both theoretically and experimentally that the natural fully synchronous parallelization can be highly inefficient. We then designed a series of alternative parallelizations by exploiting asynchrony and dynamic scheduling. Through this process we developed analytical tools to characterize the flow of information along the graph and the relationship between model parameters and the approximate sequential structure of message passing inference. In the end we arrived at the SplashBP algorithm which employs optimal tree scheduling along with advances in asynchronous dynamic scheduling to achieve an efficient scalable parallel and distributed inference algorithm. We implemented each of these algorithms in the multicore and distributed settings and demonstrated how applying the GrAD design principles can accelerate convergence, improve stability, and lead to scalable systems for probabilistic reasoning.

Motivated by our work on loopy belief propagation, in Chapter 4 we applied the GrAD design principles to the widely used Gibbs sampling algorithm from Bayesian statistics. We again demonstrated that, the natural synchronous parallelization leads to a Gibbs sampler which converges to the wrong stationary distribution.

We then applied the GrAD design principles to develop three parallel Gibbs samplers which are guaranteed to accelerate mixing while simultaneously preserving the correct stationary distribution. Through this process we also developed a set of basic analytical tools that relate properties of the graph coloring to the parallel structure of Gibbs sampling as well as the stationary distribution of the incorrect Synchronous Gibbs sampler. To address settings with tightly coupled variables we introduced the Splash Gibbs sampler which incrementally constructs multiple conditionally independent bounded treewidth blocks (Splashes) in parallel. We accelerate burn-in by dynamically growing the Splash tree structure. We implemented and evaluated the proposed samplers in the multicore setting an demonstrated strong parallel scaling and accelerated burn-in and mixing.

After validating the thesis approach by designing and implementing a collection of efficient parallel and distributed inference algorithms, we introduced GraphLab (Chapter 5) which generalizes our earlier system design efforts and specifically targets algorithms that follow the GrAD design principles. The GraphLab abstractions decomposes the program state and computation along the structure of the graph, introduces a simple sequential shared-memory model of graph computation, and supports a range of powerful dynamic scheduling primitives. As a consequence GraphLab, allows machine learning experts to compactly represent sophisticated asynchronous dynamic graph algorithms for probabilistic reasoning. The GraphLab system then executes those algorithms efficiently on multicore and distributed systems using many of the system design techniques developed in our work on probabilistic inference. We experimentally assessed the GraphLab system on a wide range of real-world applications demonstrating strong scalability and performance; further validating the thesis approach on an even wider range of machine learning and data mining applications.

While GraphLab generalized the thesis approach to a much broader class of Machine Learning and Data Mining algorithms, as we scaled to increasingly larger graphs with billions of vertices and edges we discovered a critical performance barrier. In Chapter 6, we showed that the power-law sparsity structure found in the majority of real-world graphs leads to substantial communication and storage overhead for existing graph computation systems, including GraphLab. To address this challenge we introduced PowerGraph which refined the GraphLab abstraction to further decompose vertex programs along edges in the graph and thereby enable a vertex-cut based distributed graph representation. As a consequence we were able to *leverage* the power-law structure to improve scalability and reduce communication and storage overhead. We implemented PowerGraph in the multicore and distributed setting and then ported most of the GraphLab algorithms to the PowerGraph system. Finally we theoretically and experimentally demonstrated that the PowerGraph system is capable of achieving orders-of-magnitude gains in scalability and performance over existing systems.

Throughout the thesis work, we observed several repeated patterns that go beyond the thesis statement. We now discuss some of these common patterns and their impact on our design decisions and the performance of the resulting algorithms and systems as well as their implications on future work.

7.1 Modeling Through Recurrence Relationships

Whether computing the marginal posterior of some variable or estimating the centrality of a user in a social network, the algorithms considered in this thesis defined a recurrence relationship between a vertex and its neighbors and then specified a method to resolve that recurrence relationship. In most cases, the algorithms iteratively satisfy the relationships locally and then, if necessary, repair the relationship on neighboring vertices.

The graphical recurrence relationship *simplifies* the process of designing complex models for large structured data. It is surprisingly intuitive to express complex models in terms of *local* relationships. For example, the popularity of a user in a social network can be succinctly and intuitively expressed as a simple function of the popularity of her followers. By assuming we have successfully modeled our neighbors in the graph, we can introduce a simple recurrence step to extend the model to ourselves. Moreover, by decomposing the *modeling task* to that of modeling local relationships we simplify the task of building large complex models.

The process of iteratively maintaining the graphical recurrence relationship is well suited for graph-parallel computation. Because the recurrence relationship can be expressed and maintained by only accessing the neighborhood of a vertex, we are able to factor computation along vertices exposing substantial parallelism and locality. As a consequence, we can more effectively parallelize these algorithms and represent the program state in a distributed environment.

The recurrence relationship approach to modeling and algorithm design in naturally amenable to dynamic scheduling. By defining a measure of the extent to which a recurrence is violated, we can associate priorities with each vertex. Then by focusing computational resources on the most violated recurrence relationships and propagating corrections along wavefronts we can often accelerate convergence of these algorithms. We believe that future work in the development of and application of prioritized scheduling will produce in a range of more efficient parallel and distributed algorithms.

7.2 Tradeoff between Parallelism and Convergence

By decomposing computation along vertices in a graph, graph-parallel algorithms expose substantial parallelism. In principle, we could run computation on every single vertex *simultaneously*. This realization forms the basis of existing graph-parallel systems and has contributed to the re-emergence of the bulk synchronous model of graphical computation. However, as we have demonstrated in this thesis work, such an extreme approach to parallelism leads to inefficient algorithms and in some cases inaccurate solutions.

The graphical structure of graph-parallel algorithms induces an inherent *sequential* component to their execution. At a high-level, computing the property of a vertex typically requires knowing the correct value for its neighboring vertices. This sequential dependency structure is most apparent in chain graphs and in Chapter 3 we discussed the role of chains of dependencies in the convergence of loopy belief propagation. Alternatively, in Chapter 4 we discussed how neighboring variables must be updated *sequentially* to prove algorithm correctness.

In Section 3.3 we demonstrated that the inefficiency that arises from not addressing the sequential dependencies can result in asymptotically slower convergence. However, by running computation on multiple vertices we are able to increase the computation throughput. We achieve a balance between the rate of convergence and the increased throughput due to parallelism by asynchronously prioritizing parallel resources. In addition we schedule computation along spanning trees and ensure a serializable execution that is as close to the optimal sequential execution as possible.

Similarly, in Chapter 4 we showed that the fully parallel synchronous Gibbs sampler converges to the wrong distribution. Again, by running vertex computation asynchronously and ensuring serializability we were able to obtain a parallel execution that converges to the correct posterior distribution. Finally, by moving computation along wavefronts defined by spanning trees in the graph we were able to further accelerate convergence.

By restricting our attention to serializable executions along dynamically prioritized wavefronts we are substantially reducing the amount of available parallelism. Nonetheless, for most of the problems we considered there was typically enough remaining parallelism to continue to obtain relatively strong scaling. However, as we moved to large distributed systems and very high-degree vertices (Chapter 6), there were cases where maintaining a serializable execution eliminated most of the vertex-level parallelism. For example, while executing the vertex program on the center vertex of a star graph no other vertex program may run. In these cases we were able to extract additional *edge-level* parallelism in the gather and scatter phases of the GAS decomposition. In addition, the PowerGraph abstraction enabled a substantially more efficient distributed graph representation as well as the ability to exploit *edge-level* parallelism in the distributed setting.

7.3 Approximations and Parallelism

In our work on belief propagation we revealed the connection between approximate convergence and parallelism. In particular, we showed that the limiting sequential component, defined by the parameter τ_ϵ, can be related to the accuracy of the fix-point approximation. By increasing the tolerance at convergence we can effectively increase the number of processor resources that can be used efficiently. However, the resulting algorithms did not depend on the τ_ϵ parameter and instead prioritized the use of parallel resources allowing the available resources to be used as efficiently as possible. Consequently, we expect to see a diminishing returns as parallel resources exceed the efficient level of parallelism in the model.

In our work on Gibbs sampling we showed that the limiting sequential component depends on the chromatic number of the underlying Markov Random field. However, unlike the τ_ϵ parameter in belief propagation, the chromatic number can be directly estimated from the Markov Random Field and the precise available parallelism and acceleration in mixing can be quantified. In the case of models with two colorable Markov Random Fields, we can obtain a fully synchronous approximation which introduces strong independence assumptions but also exposes the maximum available parallelism. Unfortunately we were unable to extend this analysis to Markov Random Fields with higher chromatic numbers. However, recent state-of-the-art systems [Ahmed et al., 2012, Smola and Narayanamurthy, 2010] for approximate Gibbs sampling have demonstrated the advantages of asynchronous computation and the substantial additional parallelism achieved through approximations.

7.4 Challenges of Dynamic Scheduling

Throughout our work we found dynamic scheduling to be one of the more challenging principles of the thesis methodology. To apply dynamic scheduling we had to overcome the challenges of defining an informative measures of future progress as well as address the system challenges associated with dynamically evolving computation. In addition, we had to deal with the interaction between the dynamic heuristic and the asynchronous distributed nature of its execution.

While the recursive fixed-point formulation of the vast majority of algorithms leads to a natural measure of local convergence, it is not always the optimal measure of future progress. For example in our work on belief propagation (Section 3.6.2) we found that an alternative formulation of local convergence can improve performance and help reduce asymmetric convergence. Intuitively, small changes in one vertex could potentially lead to large changes in neighboring vertices. We believe that more work into the theory of adaptive prioritization will lead to more general methods to construct prioritization.

Designing and building the data structures to manage priorities presented substantial challenges. While there is existing work in parallel priority queue design [Rao and Kumar, 1988, Shavit and Zemach, 1999] their are no widely available implementations and existing data-structures typically assume integral or uniform priorities. In our work, we found that since the priorities were directly coupled with the convergence of each vertex the range of priorities rapidly decreased making it difficult to apply more classic binning techniques. A reasonable compromise which seems to address the limitations in defining priorities as well as the scalability of scheduling data-structures is to reduce the priority to the binary *converged* or *not converged* states as was done in (Section 3.5). While a better priority definition can lead to improved performance, this simple form of *dynamic* scheduling is easy to define and implement at scale.

Whether constructing an edge-cut (e.g., as in GraphLab Section 5.7.1) or constructing a vertex-cut (e.g., as in PowerGraph Section 6.5) we are balancing work by evenly distributing the graph across machines. However, with dynamic scheduling an evenly distributed graph may lead to an imbalance in computation. In Section 3.8.1 we proposed over-partitioning as a method to improve work-balance. However, over-partitioning leads to increased communication over-head which for many applications (e.g., PageRank, Shortest Path, CoEM, Graph Coloring) is the primary bottleneck. While dynamic load balancing may provide a solution, it is difficult to predict the future work distribution given the current work distribution. We believe that form of *distributed* work stealing similar to that described by Power and Li [2010] could potentially address these issues.

7.5 Importance of High-Level Abstractions

High-level abstractions are essential to isolating the design and implementation of sophisticated algorithms from the design and implementation of complex systems. Abstractions like MapReduce or even SQL have enabled people with limited or no system design experience to design and implement specific computation patterns that efficiently leverage massive distributed systems and sophisticated data representations with minimal effort and training. In contrast, in the absence of high-level abstractions the high-performance computing community has struggled with slower development cycles and algorithms and implementations that are built around individual machines. While MPI has excelled at isolating the design and implementation of the communication layer from the algorithm and tools like Zoltan have helped isolate the challenges of graph placement, each implementation still requires substantial system design and tuning effort. For example, in our work on graphical model inference it took several years to design, implement, and tune the shared and distributed memory implementations of our algorithms using pthreads, MPI, and Metis.

However, as we demonstrated, choosing the wrong high-level abstraction can lead to substantial inefficiencies both in the abstraction assumptions as well as the requirements it imposes on the computation. For example, implementing GrAD algorithms in the MapReduce abstraction leads to excessive disk-overhead in iterative computation and forces the user to over-express parallelism in the form of synchronous computation. By identifying the key patterns that emerge across a wide range of algorithms we were able to define a new high-level abstraction which matches the needs and assumptions of the GrAD algorithms.

Our work on GraphLab and then PowerGraph demonstrated that we could both express a wide range of algorithms using a targeted abstraction and then efficiently implement that abstraction in the multicore and distributed settings. Furthermore, we observed that focused effort on the design and implementation of the abstraction lead to performance gains in the many applications built on top of the abstraction. In some cases algorithms built on top of GraphLab and PowerGraph outperformed their less optimized counterparts built using the low-level primitives from high-performance computing.

We believe that by developing high-level abstractions around common computational patterns we can isolate the challenges of algorithm and system design while enabling the construction of sophisticated algorithms that *efficiently* realize the full potential of the available system resources. As a consequence we can substantially shorten the development cycle for high-performance systems, enable systems experts to simultaneously accelerate a wide range of sophisticated algorithms, and allow domain experts to more readily leverage the immense computational resources available in parallel and distributed computing systems.

Chapter 8

Future Work

This thesis is a first step towards a larger research agenda in scalable probabilistic reasoning. The GrAD design principles and the resulting algorithms and high-level abstractions form a foundation on which to build more sophisticated algorithms, theoretical tools, and systems. In this section we describe several potential avenues for further research based on our findings and experiences.

8.1 Scalable *Online* Probabilistic Reasoning

In many real-world applications the graph structure and meta-data are rapidly evolving. For example in the context of social networks, as time progresses new users join, make friends, and change their profiles. As new data arrives and existing data is modified we would like to be able to continually update and refine our models (e.g., best estimates of users interests). We believe that the GrAD methodology will be essential in these situations since small changes to the graph should lead to localized computation that can be prioritized and executed asynchronously.

The *online* setting carries a few new challenges. First data is arriving rapidly and must be incorporated into the graph structure. The graph representations used in this work make assumptions about the ordering of vertices and their placement in distributed environments that are difficult to maintain if the structure is changing rapidly. Furthermore the semantics of graph changes are not well defined. For example, if an edge is added or deleted are both vertex programs run? When are structural changes made visible to vertex programs? Decisions about these semantics will impact the types of algorithms that may be expressed as well as the guarantees provided by the system.

The online setting typically assumes that computation runs continuously for extended periods of time requiring not only fault-tolerance but also high-availability. While we have achieved a basic form of fault tolerance through the use of both synchronously and asynchronously constructed snapshots, in the event of a machine failure we must stop all execution and restart at the last snapshot. In an online system, data is arriving continuously and must be incorporated continuously. If a machine fails, we must ensure that any recent data is not lost and that we can still incorporate new data as it continues to arrive. This form, *availability* is especially challenging to maintain in a consistent manner when there are complex data dependencies (i.e., the graph). While Pujol et al. [2010] began to address this problem, we believe that their are exciting new opportunities in the development of highly available distributed graph data-structures.

There has been only limited work in the development of online structured probabilistic reasoning. Hoffman et al. [2010] described a method to accelerate inference in the LDA model by extending the model and augmenting a variational approximation as variables are added. However, Hoffman et al. [2010] did not consider the case in which the variables are being generated from an external source and the model is defined as the fixed-point of a recursive relationship. In these cases we might like to continue to maintain end extend the model as well as the fixed-point solution. We believe that by applying reasoning similar to the τ_ϵ-approximation we might be able to show that changes in the graph have a localized effect. Finally, as the model structure changes, we need a principle method to introduce new parameters and factors and potentially relearn the parameters already in the model.

Our work on the PowerGraph abstraction demonstrated the potential for exploiting the *sparsity pattern* present in real-world problems. We also believe that a similarly exploitable *arrival pattern* is likely to be present in a wide range of real-world problems. For example, if we believe our power-law graph was derived from a preferential attachment phenomena then we might expect edges to be added to high-degree vertices disproportionately often. Alternatively, because people's online interactions are often *bursty* we might expect changes to vertex meta-data to have a similar *bursty* pattern. By tuning caching schemes and consistency models for the arrival patterns in real-world data we may be able to build more efficient online graph processing systems.

8.2 Learning a Scalable Parsimonious Structures

We have repeatedly demonstrated how the graphical structure of data dependencies affects the convergence and parallel scaling of structured probabilistic reasoning. In this work we have assumed that the structural dependencies are fixed and provided as input. In many cases this is a reasonable assumption since the model structure is coupled to relational observations like friendship, spatial proximity, or co-occurrence. However, in some cases we may not know the structure in advance or may be willing to accept a simpler structure both in pursuit of model parsimony as well as increased scalability. This observation leads to the natural questions: can we *regularize for scalability* and thereby efficiently *learn scalable structures*?

There are several challenges to scalable structure learning. First, in general structure learning is computationally expensive and typically requires parameter learning and inference as a subroutines. Second general structure learning algorithms, for obvious reasons, lack the graphical decomposition that was central to the GrAD methodology and may require reasoning about arbitrary subsets of the data. Third, structure learning can require large amounts of data to infer reasonable structures. If we must assess conditional independence between groups of variables we will need many joint observations of subsets of variables. Finally, we need to define a metric which effectively captures the notion of a *scalable* structure.

To overcome some of the computational and data efficiency challenges we might pursue structure search algorithms that incrementally build upon simpler structures. This would ensure that we are never required to do inference or parameter learning on a potentially unscalable or intractable structure. Furthermore, by building on earlier structures we may be able to *reuse* previous inference estimates (e.g., belief propagation messages) to warm start the inference process. Furthermore, by building upon simpler structures we may be able to leverage some of the distributed representation of the previous structure.

In large-scale real-world structure search there is often considerable domain knowledge about which variables might be related and which features may be informative. For example in image processing we might believe that nearby pixels have similar values but we may not know the exact local structure. Alternatively, in the social networking domain we may assume that the model structure is a *sub-graph* of the overall social network. In these cases we may be able to restrict our attention to a smaller set of possible structures and in some cases sub-structures enabling a graphical decomposition of the search process. Furthermore, by restricting the search space we may be able to learn structures using fewer training examples.

The remaining and potentially most interesting challenge is the definition of a metric which captures the notion of a *scalable* structure. Moreover, what defines a scalable structure? The thesis work suggests several potential criterion. If we desire a serializable execution, we would like to minimize the chromatic number of the underlying graph. Alternatively, if we are most worried about communication and work balance, we might like to identify a structure which minimizes cost of the corresponding vertex cut. Finally, if we want to accelerate the convergence of the underlying inference algorithm than we might want to find a structure which has many *short* paths between variables (e.g., expander graphs). We believe their are exciting opportunities in the study of regularization for scalability and its impact on model and system performance.

8.3 Generalizing Prioritized Computation

One of the key principles in the GrAD methodology is the dynamic prioritization of computation. However, defining informative metrics for prioritization can be challenging. For example, in our work on belief propagation and Gibbs sampling, we spent a considerable effort to define informative heuristics on which to prioritize computation. If we assume our algorithm is a contraction mapping we can use the difference between subsequent computation steps as a priority metric which greedily accelerates convergence. However, many algorithms are not a contraction mapping and so the priority metric is less obvious. For example, in our work on CoEM we struggled to identify an informative definition of priorities. In these cases we typically resorted to binary priorities (e.g., converged or not converged). Nonetheless, we conjecture that the study of prioritized could lead to accelerated convergence for a wide range of algorithms.

8.4 Dynamic Work Balancing

Dynamic computation is one of the key principles of the GrAD design methodology which accelerates convergence and eliminates wasted computation by focusing computational resources where they can make the most progress. Just as with adaptive mesh refinement in high-performance computing scenarios, adaptive prioritization focuses computation on sub-regions of the original problem. However, if the problem was originally evenly distributed over the cluster, it is common that some machines may actually have disproportionately more work.

In Section 3.8.1 we introduced over-partitioning as a randomized strategy to address the challenges of work imbalance due to dynamic computation in the distributed setting. While over-partitioning is appealing in its simplicity and can be shown to improve work balance, it increases communication throughout the execution of the system. Alternatively, we could consider occasionally repartitioning the graph, as is commonly done with mesh refinement in high-performance computing. However, we found in our work on belief propagation, the dynamic computation can be relatively unpredictable. Instead, we believe that *work* *stealing* may be more effective in these situations. Work stealing has the advantage of moving work only when imbalance arrises and can be used to address imbalance in computation as well as lagging processor resources. Power and Li [2010] developed a block based work-stealing mechanism which works well in distributed environments to address both imbalance in the processing time of blocks as well as slow processors.

8.5 Declarative Abstractions for Recurrence Resolution

Many of the GrAD algorithms effectively encode graph computation by modeling a recursive relationship between a vertex and its neighboring vertices. The GraphLab and PowerGraph abstractions imperatively express how to enforce and repair these recursive relationship and prioritize corrections to neighboring vertices. In most cases the process of repairing these recurrence relationships is to directly recompute the recurrence locally and trigger any neighbors to be recomputed if their recurrence is violated. As a consequence, it may be possible to express these algorithms (or even models) at a higher more *declarative* level freeing the system to determine how and when to prioritize computation, enabling composition of multiple recurrence relations, and simplifying the transition to online computation.

Instead of expressing the algorithm to compute and preserve a local recurrence relationship, a declarative formulation of GrAD algorithms would instead specify the value of a vertex attribute as a function of its neighborhood. To assess approximation convergence (e.g., τ_ϵ-approximation) and enable prioritization we may also want to introduce a measure of recurrence violation and a acceptable convergence threshold ϵ. For example, the PageRank algorithm could be expressed compactly as just the PageRank equation, a measure of change, and a termination condition:

$$\begin{aligned} \mathbf{R}(v) &\leftarrow \frac{\alpha}{n} + (1-\alpha) \sum_{u \text{ links to } v} w_{u,v} \times \mathbf{R}(u) \\ \text{Change}(\mathbf{R}(v)^{\text{old}}, \mathbf{R}(v)^{\text{new}}) &:= \left| \mathbf{R}(v)^{\text{old}} - \mathbf{R}(v)^{\text{new}} \right| \\ \epsilon &:= 0.005 \end{aligned}$$

By expressing PageRank as recurrence relationship, a measure of change, and a termination condition, we free the system and execution environment to reason about scheduling and dynamic computation. For example, the system could optimize over different scheduling primitives. In addition, the recurrence formulation can automatically be translated into an *online* algorithm. For example, if the graph structure were to change then the system could directly assess if the recurrence had been violated and then choose to trigger future computation as needed. Finally, we could introduce addition recurrence relationships that depend on the values of other recurrence relationships. For example, we might associated the marketing value of a user with the PageRank of that user as well as the marketing values of neighboring users.

8.6 The GrAD Methodology and General Graph Computation

The GrAD methodology could potentially be applied more broadly to general graph computation. The asynchronous prioritized approach to graphical model inference as well as many of the analysis techniques could be naturally applied to discrete simulations and the evaluations of large-scale partial differential equations. In addition the GraphLab and PowerGraph abstractions could also be used to simplify the design and implementation of graphical computation.

Chapter 9

Summary of Notation and Terminology

While we have attempted to keep each chapter as self-contained and use consistent terminology and notation, it can still be difficult to track the various terminology and notation through this document. In this chapter we provide a brief summary.

9.1 Terminology

GrAD Methodology: The The **Gr**aphical, **A**synchronous, **D**ynamic, (**GrAD**), design methodology is composed of the following three basic principles:

1. **Graphical**: Explicitly represent statistical and computational dependencies as a sparse graph. The sparse graphical representation enables the parallel decomposition of algorithms and exposes the data locality needed for efficient distributed computation.

2. **Asynchronous**: Allow computation to proceed asynchronously as dependencies are satisfied obeying the dependency structure encoded in the graph. By allowing computation to run asynchronously as dependencies are satisfied we can reduce wasted computation and build more efficient parallel and distributed systems. By obeying the dependency structure we can obtain provably correct asynchronous algorithms.

3. **Dynamic**: Adaptively identify and prioritize sub-problems that when solved make the most progress towards the solution. By first identifying and solving the key sub-problems we can reduce the overall work and more effectively use computational resources.

Asynchronous Computation: Computation that runs as new data arrives and resources become available without a single centralized clock or source of coordination. Asynchronous algorithms typically use the most recent information available and communicate changes.

Synchronous Computation: All computation runs simultaneously in phases that are coordinated by a single clock. Typically communication only occurs between phases and therefore computation operates on the output of the previous phase.

Factor Graph: A representation of a graphical model as a bipartite graph with variables on one side connected to the clique factors on the other side.

Markov Random Field: The Markov random field of a graphical model is the graph over the variables in which each variable is connected to all variables in its Markov blanket.

Markov Blanket: The Markov blanket of a variable are the set of variables which when observed separate render the variable *independent* from the rest of the variables in the model.

9.2 Notation

Joint Probability P $(X_1, \ldots, X_n)$: The join probability function for the variables X_1 to X_n describes the probability of particular assignment to all random variables. In this work we focus on discrete random variables but many of the techniques extend to continuous random variables.

Number of variables/vertices $]numvars$**:** We use n to represent the number of variables in the model (or vertices in the graph).

Number of processors p: We denote the number of individual computational units (really threads) by p.

Conditional Probability P $(X_i \,|\, X_j)$: The conditional probability functions describes the probability of X_i given (denoted by ' $|$ ') a particular value for X_j.

Model parameters θ: We use θ to denote the model parameters of a graphical model. The model parameters along with the graphical structure define the joint probability function.

Clique set $\mathcal{F}$: The set $\mathbf{A} \in \mathcal{F}$ consists of all maximal cliques $\mathbf{A}$ in the Markov random field.

Model Parameters θ: The model parameters encode the strength and form of the relationships between connected variables in a graphical models.

Clique factors $f_{\mathbf{A}}(\mathbf{x}_{\mathbf{A}} \,|\, \theta)$: The clique factors in a factor graph encode the local relationship between the variables in the set $\mathbf{A} \subset \{1, \ldots, n\}$ as a positive function which depends on the parameters θ.

Partition Function $Z(\theta)$: The partition function encodes the normalizing constant for the joint probability function and depends only on the model parameters θ.

The sample space $\mathcal{X}_i$ of variable X_i: The sample space of a variable is the set of possible value the variable can take.

Neighbors $\mathcal{N}_i$: The set of neighbors of vertex i is denoted by $\mathcal{N}_i$.

Belief Propagation messages $m_{X_i \to f_{\mathbf{A}}}(x_i)$ and $m_{f_{\mathbf{A}} \to X_i}(x_i)$ The belief propagation messages are iteratively refined in the belief propagation update equations. The messages can be related to the Lagrange multipliers in the Bethe approximation or in tree graphical models the messages encode the contribution to the variable marginal after eliminating all variables in the originating subtree.

Belief P $(\mathbf{X}_{\mathbf{A}} = \mathbf{x}_{\mathbf{A}}) \approx b_{\mathbf{X}_{\mathbf{A}}}(\mathbf{x}_{\mathbf{A}})$: The posterior marginals (beliefs) estimated by the belief propagation algorithm.

τ_ϵ-approximation: The τ_ϵ-approximation accounts for factor dependent serial structure of approximate inference. The number of iterations (or distance) τ_ϵ is the minimum number of iterations (or distance that a message must travel) to achieve an ϵ level approximation of the message at convergence.

Expectation E $[X]$: The expected value of the random variable X.

Transition kernel $K(X' \mid X)$: The transition kernel of a Gibbs sampler assigns the probability of transition from state X to state X'.

Junction Tree $(\mathcal{C}, E) = \mathbf{J}_{\mathcal{S}}$: The set of vertices $\mathbf{C} \in \mathcal{C}$ is called a clique and is a subset of the vertices (i.e., $\mathbf{C} \subseteq \mathcal{S}$) in the Splash $\mathcal{S}$. The cliques satisfy the constraint that for every factor domain $\mathbf{A} \in \mathcal{F}$ there exists a clique $\mathbf{C} \in \mathcal{C}$ such that $\mathbf{A} \cap \mathcal{S} \subseteq \mathbf{C}$. The edges E of the junction tree satisfy the running intersection property (RIP) which ensures that all cliques sharing a common variable form a connected tree.

PageRank Reset Probability γ: In the random surfer model there is a small probability, typically $\gamma = 0.15$ of jumping to a random webpage.

Vertex Data D_u and edge data $D_{u \to v}$: The GraphLab and PowerGraph abstractions associate arbitrary user defined data with each vertex and directed edge in the data graph.

Accumulator a_u: The Gather phase of the PowerGraph abstraction accumulates the result of the gather function in the accumulator a_u for the vertex u. The accumulator a_u is then saved for that vertex and updated using the scatter function.

Degree $\mathbf{D}[v]$ of vertex v: The degree ($|\mathcal{N}_v|$) is denoted by $\mathbf{D}[v]$.

Power-Law constant α: The skewness of a power-law degree distribution is determined by the power-law constant which is typically $\alpha \approx 2$.

Harmonic function $\mathbf{h}_x(\alpha)$: The harmonic function is the normalizer of the discrete bounded Zipf distribution ($\mathbf{h}_x(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$).

Assignment function $A(v)$: The partition assignment function $A(v)$ maps a vertex (or edge) to the set of machines on which it is located.

Bibliography

Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, 2006.

Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

R. Albert, H. Jeong, and A. L. Barabási. Error and attack tolerance of complex networks. In *Nature*, volume 406, pages 378—482, 2000.

Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008. ISSN 0360-0300.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, Al Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010. ISSN 0001-0782.

K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

A. Asuncion, P. Smyth, and M. Welling. Asynchronous distributed learning of topic models. In *NIPS*, 2008.

A. Barbu and S. Zhu. Generalizing swendsen-wang to sampling arbitrary posterior probabilities. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8), 2005.

D. Baron, S. Sarvotham, and R. G. Baraniuk. Bayesian compressive sensing via belief propagation. *Trans. Sig. Proc.*, 58(1): 269–280, 2010.

D. Batra, A. Kowdle, D. Parikh, L. Jiebo, and C. Tsuhan. iCoseg: Interactive co-segmentation with intelligent scribble guidance. In *CVPR*, pages 3169 –3176, 2010.

D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1989.

D. Bickson. *Gaussian Belief Propagation: Theory and Application*. PhD thesis, The Hebrew University of Jerusalem, 2008.

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003. ISSN 1532-4435.

Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596, 2011.

Ilaria Bordino, Paolo Boldi, Debora Donato, Massimo Santini, and Sebastiano Vigna. Temporal evolution of the uk web. In *ICDM Workshops*, pages 909–918, 2008.

Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767921. URL `http://dx.doi.org/10.1109/ICDE.2011.5767921`.

Aydin Buluç and John R. Gilbert. The combinatorial blas: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

Umit Catalyurek and Cevdet Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In

IRREGULAR, pages 75–86, 1996.

S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989. ISSN 1041-4347. doi: 10.1109/69.43410. URL `http://dx.doi.org/10.1109/69.43410`.

H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP '11*. ACM, 2011.

K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984. ISSN 0164-0925. doi: 10.1145/1780.1804. URL `http://doi.acm.org/10.1145/1780.1804`.

K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985. ISSN 0734-2071.

Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large graph processing in the cloud. In *SIGMOD*, pages 1123–1126, 2010. ISBN 978-1-4503-0032-2.

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288. 2006.

G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42: 393–405, 1990.

V.A. Crupi, S.K. Das, and M.C. Pinotti. Parallel and distributed meldable priority queues based on binomial heaps. In *Parallel Processing, International Conference on*, volume 1. IEEE Computer Society, 1996.

A. Darwiche, R. Dechter, A. Choi, V. Gogate, and L. Otten. Uai'08 workshop: Evaluating and disseminating probabilistic reasoning systems, 2008. `http://graphmod.ics.uci.edu/uai08/`.

J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *IPDPS*, 2006.

Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

P. Domingos. Uw-cse mlns, 2009. URL `alchemy.cs.washington.edu/mlns/cora`.

F. Doshi-Velez, D. Knowles, S. Mohamed, and Z. Ghahramani. Large scale nonparametric bayesian inference: Data parallelisation in the indian buffet process. In *NIPS 22*, 2009.

J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31:1343–1354, 1988.

B Efron, T Hastie, I M Johnstone, and Robert Tibshirani. Least angle regression. *Annals of Statistics*, 32(2):407–499, 2004.

J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC*. ACM, 2010.

G. Elidan, I. McGraw, and D. Koller. Residual Belief Propagation: Informed scheduling for asynchronous message passing. In *UAI*, pages 165–173, 2006.

M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM Computer Communication Review*, 29(4):251–262, 1999.

P. A. Ferrari, A. Frigessi, and R. H. Schonmann. Convergence of some partially parallel gibbs samplers with annealing. *The Annals of Applied Probability*, 3(1), 1993.

Wenjiang J. Fu. Penalized regressions: The bridge versus the lasso. *Journal of Computational and Graphical Statistics*, 7(3): 397–416, 1998.

S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the bayesian restoration of images. In *PAMI*, 1984.

J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *AISTATS*, volume 15, pages 324–332, 2011.

H.P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik. Parallel support vector machines: The cascade SVM. In *NIPS*, pages 521–528, 2004.

D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *POOSC*, 2005.

Firas Hamze and Nando de Freitas. From fields to trees. In *UAI*, 2004.

B. Hendrickson and R. Leland. The chaco user's guide, version 2.0. Tech. Rep. SAND94-2692, Sandia National Labs, Albuquerque, NM, October 1994.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A common substrate for cluster computing. In *HotCloud*, 2009.

Matthew Hoffman, David Blei, and Francis Bach. Online learning for latent dirichlet allocation. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 856–864. 2010.

Thomas Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, pages 50–57, 1999.

J. Huang, M. Chavira, and A. Darwiche. Solving MAP exactly by searching on compiled arithmetic circuits. In *AAAI'06*, 2006.

A.T. Ihler, J.W. Fischer III, and A.S. Willsky. Loopy belief propagation: Convergence and effects of message errors. *J. Mach. Learn. Res.*, 6:905–936, 2005.

M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

A. Jaimovich, G. Elidan, H. Margalit, and N. Friedman. Towards an integrated protein–protein interaction network: A relational markov network approach. *Journal of Computational Biology*, 13(2):145–164, 2006.

C. S. Jensen and A. Kong. Blocking gibbs sampling for linkage analysis in large pedigrees with many loops. In *American Journal of Human Genetics*, 1996.

M. Jiayuan, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *IPDPS*, pages 1 –12, May 2009. doi: 10.1109/IPDPS.2009.5160991.

R. Jones. *Learning to Extract Entities from Labeled and Unlabeled Text*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 2005.

U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, pages 229 –238, 2009.

U Kang, D.H. Chau, and C. Faloutsos. Inference of beliefs on billion-scale graphs. *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

Seung-Jean Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An interior-point method for large-scale $\ell 1$-regularized least squares. *IEEE J SEL TOP SIGNAL PROC*, 1, 2007.

S. Kogan, D.Levin, B.R. Routledge, J.S. Sagi, and N. A. Smith. Predicting risk from financial reports with regression. In *Proceedings of the North American Association for Computational Linguistics Human Language Technologies Conference*, pages 272–280, 2009.

D. Koller and N. Friedman. *Probabilistic Graphical Models*. MIT Press, 2009.

A.V. Kozlov and J.P. Singh. A parallel lauritzen-spiegelhalter algorithm for probabilistic inference. In *Supercomputing '94. Proceedings*, pages 320 –329, November 1994.

M. Kubale. *Graph Colorings*. American Mathematical Society, 2004.

M. Kuss and C. E. Rasmussen. Assessing approximate inference for binary gaussian process classification. *J. Mach. Learn. Res.*, 6, 2005.

H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

X. Y. Lan, S. Roth, D. P. Huttenlocher, and M. J. Black. Efficient belief propagation with learned higher-order markov random fields. In *ECCV'06*, 2006.

K. Lang. Finding good nearly balanced cuts in power law graphs. Technical Report YRL-2004-036, Yahoo! Research Labs, Pasadena, CA, Nov. 2004.

Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, pages 85–94, 2011.

J. Leskovec. Stanford large network dataset collection. http://snap.stanford.edu/data/index.html, 2011.

J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1), mar 2007.

J. Leskovec, K. J. Lang, A. Dasgupta, , and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2008.

R. A. Levine and G. Casella. Optimizing random scan Gibbs samplers. *J. Multivar. Anal.*, 97(10):2071–2100, 2006.

S. Z. Li. *Markov random field modeling in computer vision*. Springer-Verlag, London, UK, 1995. ISBN 4-431-70145-1.

W. G. Macready, A. G. Siapas, and S. A. Kauffman. Criticality and parallelism in combinatorial optimization. *Science*, 271: 271–56, 1995.

G. Malewicz, M. H. Austern, A. J.C Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *SYSTSOFT*, 43, 1998.

F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

A. Mendiburu, R. Santana, J.A. Lozano, and E. Bengoetxea. A parallel framework for loopy belief propagation. In *GECCO'07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, 2007.

G. Miller and J. Reif. Parallel tree contraction and its applications. In *In Proceedings of the 26th IEEE Annual Symposium on Foundations of Computer Science*, pages 478–489, 1985.

T. P. Minka. *A family of algorithms for approximate bayesian inference*. PhD thesis, MIT, 2001. AAI0803033.

J. Misra. Detecting termination of distributed computations using markers. In *PODC*, pages 290–294, 1983.

J.M. Mooij and H.J. Kappen. Sufficient conditions for convergence of the Sum-Product algorithm. *ITIT*, pages 4422–4437, 2007.

R. Nallapati, W. Cohen, and J. Lafferty. Parallelized variational EM for latent Dirichlet allocation: An experimental evaluation of speed and scalability. In *ICDMW*, pages 349–354, 2007.

R.M. Neal and G.E. Hinton. A view of the EM algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. 1998.

Neo4j. The world's leading graph database. `http://neo4j.org`, 2011.

D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed inference for latent dirichlet allocation. In *NIPS*, pages 1081–1088, 2007.

K. Nigam and R. Ghani. Analyzing the effectiveness and applicability of co-training. In *CIKM*, 2000.

L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

B. Panda, J.S. Herbach, S. Basu, and R.J. Bayardo. Planet: massively parallel learning of tree ensembles with MapReduce. *Proc. VLDB Endow.*, 2(2):1426–1437, 2009.

I. Parberry. Load sharing with parallel priority queues. *J. Comput. Syst. Sci*, 50(1):64–73, 1995.

R. Pearce, M. Gokhale, and N.M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SC*, pages 1–11, 2010.

J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN Europe*, pages 493–498, 1996.

David M. Pennock. Logarithmic time parallel Bayesian inference. In *Proc. 14th Conf. Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 1998.

R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. In *SIGCOMM*, pages 375–386, 2010.

A. Ranganathan, M. Kaess, and F. Dellaert. Loopy sam. In *IJCAI'07*, 2007.

V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Trans. Comput.*, 37(12):1657–1665, 1988.

Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2), 2006.

D. Roth. On the hardness of approximate reasoning. In *ijcai93*, pages 613–618, 1993.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

P. Sanders. Randomized static load balancing for tree-shaped computations. In *Workshop on Parallel Processing*, pages 58–69. TR Universitat Clausthal, 1994.

P. Sanders. On the competitive analysis of randomized static load balancing, 1996.

Peter Sanders. Randomized priority queues for fast parallel access. *J. Parallel Distrib. Comput*, 49(1):86–97, 1998.

E. Shapiro. Systolic programming: a paradigm of parallel processing. *Concurrent Prolog*, 1988.

N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *In Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC), Atlanta*, pages 113–122, 1999.

Parag Singla and Pedro Domingos. Lifted first-order belief propagation. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*, pages 1094–1099. AAAI Press, 2008. ISBN 978-1-57735-368-3. URL `http://portal.acm.org/citation.cfm?id=1620163.1620242`.

A. J. Smola and S. Narayanamurthy. An Architecture for Parallel Topic Models. *PVLDB*, 3(1):703–710, 2010.

Ameet Soni, Craig Bingman, and Jude Shavlik. Guiding belief propagation using domain knowledge for protein-structure determination. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, BCB '10, pages 285–294, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0438-2. doi: 10.1145/1854776.1854816. URL `http://doi.acm.org/10.1145/1854776.1854816`.

Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. Technical Report MSR-TR-2011-121, Microsoft Research, November 2011.

Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. In *ISWC*, pages 764–780, 2010.

J. Sun, H. Y. Shum, and N. N. Zheng. Stereo matching using belief propagation. In *ECCV'02*, 2002.

Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

Daniel Tarlow, Dhruv Batra, Pushmeet Kohli, and Vladimir Kolmogorov. Dynamic tree block coordinate ascent. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 113–120, New York, NY, USA, June 2011. ACM. ISBN 978-1-4503-0619-5.

Ben Taskar, Vassil Chatalbashev, and Daphne Koller. Learning associative markov networks. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 102, New York, NY, USA, 2004a. ACM. ISBN 1-58113-828-5. doi: http://doi.acm.org/10.1145/1015330.1015444.

Ben Taskar, Ming-Fai Wong, Pieter Abbeel, and Daphne Koller. Link prediction in relational data. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004b.

S. Tatikonda and M. I. Jordan. Loopy belief propogation and gibbs measures. In *UAI'02*, 2002.

R. Tibshirani. Regression shrinkage and selection via the lasso. *J ROY STAT SOC B*, 58:267–288, 1996.

Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X.

Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. ISSN 0001-0782. doi: 10.1145/1435417.1435432. URL `http://doi.acm.org/10.1145/1435417.1435432`.

M. Wainwright, T. Jaakkola, and A.S. Willsky. Tree-based reparameterization for approximate estimation on graphs with cycles. In *NIPS*, 2001.

M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky. Tree-based reparameterization framework for analysis of sum-product and related algorithms. *IEEE Trans. Inf. Theor.*, 49(5):1120–1146, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.2003.810642. URL `http://dx.doi.org/10.1109/TIT.2003.810642`.

J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. In *ICML*. ACM, 2008.

Yinglong Xia and Viktor K. Prasanna. Parallel exact inference on the cell broadband engine processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 58:1–58:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL `http://portal.acm.org/citation.cfm?id=1413370.1413429`.

Yinglong Xia and Viktor K. Prasanna. Scalable node-level computation kernels for parallel exact inference. *IEEE Trans. Comput.*, 59:103–115, January 2010. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.2009.106. URL `http://dx.doi.org/10.1109/TC.2009.106`.

Yinglong Xia, Xiaojun Feng, and Viktor K. Prasanna. Parallel evidence propagation on multicore processors. In *Proceedings of the 10th International Conference on Parallel Computing Technologies*, PaCT '09, pages 377–391, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03274-5. doi: http://dx.doi.org/10.1007/978-3-642-03275-2_37. URL `http://dx.doi.org/10.1007/978-3-642-03275-2_37`.

E. P. Xing, M. I. Jordan, and S. Russell. A Generalized Mean Field Algorithm for Variational Inference in Exponential Families. In *UAI '03*, pages 583–559, 2003.

F. Yan, N. Xu, and Y. Qi. Parallel inference for latent dirichlet allocation on graphics processing units. In *NIPS*, 2009.

C. Yanover and Y. Weiss. Approximate inference and protein folding. In *NIPS*, pages 84–86, 2002.

C. Yanover, O. Schueler-Furman, and Y. Weiss. Minimizing and learning energy functions for side-chain prediction. *J Comput Biol*, pages 381–395, 2007.

J. Ye, J. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM*. ACM, 2009.

J.S. Yedidia, W.T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. In *Exploring artificial intelligence in the new millennium*, pages 239–269, 2003.

John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17:530–531, 1974.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: a distributed framework for prioritized iterative computations. In *SOCC*, pages 13:1–13:14, 2011.

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM*, pages 337–348, 2008.

ML
MACHINE LEARNING
DEPARTMENT

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Carnegie Mellon